

Appendix B

A Dictionary of VB .NET

Both experienced Visual Basic programmers and those new to the language can make good use of this appendix.

Showing Traditional VB Commands and Their .NET Equivalents

This appendix — which is the size of a small book — contains an alphabetized list of VB .NET features, including new techniques, differences between VB 6 (and earlier versions) and VB .NET, and the best ways to use VB .NET. I wrote this appendix during the past 18 months while I was learning VB .NET. I knew traditional VB quite well, but making the adjustment to VB .NET took time. VB .NET has many new features, and some of them take getting used to (such as streaming I/O, printing, file access, and so on). But VB .NET gives you much greater, finer control. After you get to know VB .NET capabilities (such as the ArrayList), you see how they are superior to traditional VB features.

Throughout this appendix, you can find code examples that you can try out to get a deeper understanding of the many facets of VB .NET.

VB .NET 2003



If you have upgraded to VB .NET 2003, you will find that there are only two significant changes to the language in this version: the bit shifting operators << and >> have been added, and you can initialize a loop counter. You can declare a loop counter variable within the line that starts the loop, like this:

```
For i As Integer = 0 To 10
```

instead of the traditional style:

```
Dim i as Integer  
For i = 0 To 10
```

However, the VB .NET 2003 Help system is significantly improved, offering many more tested, working code samples, and additional narrative descriptions showing you how to accomplish various jobs. The IDE is also better, particularly because it now separates procedures and other zones visually with a thin, gray line. This makes it easier to scroll through your source code and locate what you're looking for.

Using Classic Functions

Many classic VB built-in functions are still available in VB .NET, held in wrappers. Generally, there is no difference between the classic function's behavior and the new .NET wrapper version. For example, if you want to convert a variable to a string-type variable, you can either use the classic `CStr` function or the new VB .NET `ToString` method.

The VB .NET equivalents of these legacy wrappers, though, are generally methods. For instance, the classic VB `Len` function tells you how long a string variable is: `Len(string)`.

But in VB .NET, the optional replacement is the `Length` method of the `string` object: `string.Length`.

For the VB .NET equivalents of all the string functions (`Mid`, `Instr`, and so on), see "String Functions Are Now Methods," later in this appendix.

Sometimes, if you want to use a classic function, you must add this line at the very top of your code window:

```
Imports Microsoft.VisualBasic
```

This is the compatibility namespace, and it includes many traditional VB commands. However, this namespace isn't required when you use wrapped functions such as `Len`. (See "Imports" in this appendix.) The main point is that if you try to use a classic function and you get an error message from VB .NET, you should add the `Imports` line to see whether that cures the problem.

To see the classic VB commands, press F2 in the VB .NET editor to open the Object Browser utility. You may have to press Ctrl+Alt+J instead.

Click the + next to Microsoft Visual Basic .NET Runtime to open its tree list, and then click the + next to Microsoft Visual Basic to open its list of general categories. Keep drilling down in these tree lists until you locate the classic VB command that you are interested in. Then click the Member in the right pane of the Object browser to see the proper syntax. For example, if you click the `FileSystem` object in the left pane and then click the `CurDir` member in the right pane, you see this syntax:

```
Public Shared Function CurDir() As String
    Member of Microsoft.VisualBasic.FileSystem
```

With this syntax, you can then extrapolate the actual code that you would need to write to find out the current directory (CurDir's job):

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    Dim s As String
    s = CurDir()
    MsgBox(s)

End Sub
```

Run this code, and VB .NET will display the current directory.

+ =

(See "Optional Operation Shorthand")

AND, OR, XOR, and NOT

Before VB .NET

And, Or, Not, and XOr worked on the bit level.

VB .NET

Early in the development of VB .NET, there were plans to make the commands `And`, `Or`, `Not`, and `XOr` logical rather than bitwise operators. The plan was to include a new set of operators to deal with bitwise operations. This plan was abandoned and these four operators now work the same way that they always have within VB. Traditional operator precedence is also retained.

Two new operators, `AndAlso` and `OrElse`, can be used if you need to make a logical comparison. (These new operators work only on Boolean types.)

These four operators now are "overloaded," which means they can work both ways, depending on the parameters passed to them. They perform a logical exclusion operation if you pass two Boolean expressions or a bitwise exclusion if you pass two numeric expressions.

Traditional operator precedence is also retained.

If you are using VB .NET 2003, bit shifting operators have been added to the language: << and >>.

App Object

In traditional VB, you could use the `App` object to find out information about the currently running app, such as `App.EXENAME`. However, in VB .NET, you get the `.exe` name this way:

```
Dim s As String
s = Environment.CommandLine 'running program
MsgBox (s)
```

For additional data that you can get from the `Environment` object, see “Directory (Current).”

Array Declarations

Other .NET languages declare array sizes differently than VB. In other languages, the following declaration results in an array with 10 elements indexed from `X(0)` up to `X(9)`:

```
Dim X(10)
```

In other words, the number in parentheses describes the total number of elements in the array.

However, that same line of code in VB has always resulted in an array with 11 elements indexed from `X(0)` up to `X(10)`. In traditional VB, the number in parentheses represents the highest index number that can be used with this array, not the total number of elements.

Early in the development of the .NET languages, VB .NET was going to be made to conform to the rest of the .NET languages in this issue, but that attempt was abandoned. VB .NET will retain the traditional VB approach to array declaration. For more on this topic, see “Zero Index Controversy” and the various source code examples for which the `Dim` command is used.

For information on how you can simultaneously declare and initialize an array, see “Dim and ReDim: New Rules.”

Array Dimensioning Changes

See “Zero Index Controversy.”

Array of Objects

In VB .NET, you can create an array of objects. The trick is to first declare an array object variable and then instantiate each object in the array. The following example illustrates how to create an array of seven objects:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    (Windows Form Designer generated code)
    Dim arrRecipe(6) As recipe 'create the array object variable
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
        As System.EventArgs) Handles MyBase.Load
        Dim i As Integer
        'instantiate each member of the array:
        For i = 0 To 6
            arrRecipe(i) = New recipe()
        Next
        ' set the two properties of one of the array members
        arrRecipe(0).Title = "MyZeroth"
        arrRecipe(0).Description = "MyZeroth recipe goes like
            this"
    End Sub
End Class

Public Class recipe
    Private _Title As String
    Private _Description As String

    Public Property Title() As String
        Get
            Return _Title
        End Get
        Set(ByVal Value As String)
            _Title = Value
        End Set
    End Property
End Class
```

```
Public Property Description() As String
    Get
        Return _Description
    End Get
    Set(ByVal Value As String)
        _Description = Value
    End Set
End Property

End Class
```

ArrayList (Replacing the Array Command)

The new VB .NET `ArrayList` is a powerful tool. You'll want to take advantage of the many features of this new object. Familiarize yourself with it if you expect to ever need to manipulate arrays in your programming. For one thing, it can dynamically resize itself, so you don't have to resort to `ReDim` and other tortured techniques.

Before VB .NET

No real equivalent of the `ArrayList` existed before VB .NET, but there was the rather simple `Array`.

For most of the 10-year VB history, there was no function that created an array and then allowed you to add some data to it during runtime. Then, in VB 6, the `Array` function was added to do just that. (Those of you familiar with early, 1980s forms of BASIC will recall the similar `DATA` statement that was used to insert items of data into an executing program.)

You could use the `Array` command in VB 6 to stuff some data into an array, like this:

```
Private Sub Form_Load()

    Arx = Array("key", "Name", "Address")
    MsgBox (Arx(2))

End Sub
```

VB .NET

The `Array` function has been deleted from the language, presumably because it returns a variant variable type, and variants are disallowed in the .NET framework.

The new `ArrayList` object, however, does what the `Array` function used to do, and much, much more.

Here's a VB .NET equivalent to the old `Array` command:

```
Dim MyArray as new ArrayList

myArray.Add ("key")
myArray.Add ("Name")
myArray.Add ("Address")
MsgBox (MyArray(2))
```

You can also stuff data into an array during its declaration using the following strange method in VB .NET:

```
Dim Month() As Integer = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
                          12}

MsgBox (month(4))
```

Running this code results in a `MsgBox` displaying 5 (the “fourth” index item in the array, because arrays start with a 0 index).

This is strange code because Visual Basic has never in its history used the brace characters `{` and `}`. The C languages, of course, use braces everywhere.

The ArrayList powerhouse

But back to our regular programming. The new `ArrayList` is excellent.

To see it in action, start a new VB .NET Windows-style project and put a `ListBox` on the form. Then add four `Buttons` to the form, as well. Each `Button` will do a different `ArrayList` trick.

First, create a variable (shown in bold) near the top of the form:

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Dim arrList As New ArrayList()
```

Then type in the following procedures:

```
Public Function addOne() As String

    ListBox1.Items.Clear()
    ListBox1.Items.AddRange(arrList.ToArray)
    Me.Text = arrList.Count.ToString

End Function

Private Sub Button1_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button1.Click

Dim s As String = InputBox("Please type something")
If s <> "" Then
    arrList.Add(s)
    addOne()
End If
End Sub

Private Sub Button2_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button2.Click
    arrList = ArrayList.Repeat("Sandy", 12)
    addOne()
End Sub

Private Sub Button3_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button3.Click

    arrList.Reverse()
    addOne()
End Sub

Private Sub Button4_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button4.Click

Dim s As String = InputBox("Search ArrayList for ...?")

    If arrList.Contains(s) Then
        MsgBox("Yes, " + s + " was in the array list.")
    End If
End Sub
```

The first function clears out the `ListBox`, displays the array in the `ListBox`, and then displays the count (the total number of items in the `ArrayList`) in the title bar of the form.

When you run this program, `Button1` illustrates how to add a single item to an `ArrayList`. `Button2` shows how to fill an `ArrayList` with any number of duplicate objects. `Button3` reverses the order of the objects in the `ArrayList`, and `Button4` searches the `ArrayList`.

An `ArrayList` can do many more things. Search Help for “`ArrayList`” to see all of its methods and properties.

Array Search and Sort Methods

You can now have arrays search themselves or sort themselves. Here’s an example showing how to use both the sort and search methods of the `Array` object. The syntax for these two methods is as follows:

```
Array.Sort(myArray)
```

And

```
anIndex = Array.BinarySearch(myArray, "trim")
```

To see this feature in action, put a `Button` and a `TextBox` control on a form and change the `TextBox`’s `MultiLine` property to `True`. Then enter the following code:

```
Private Sub Button1_Click(ByVal sender As System.Object,  
    ByVal e As System.EventArgs) Handles Button2.Click  
  
    Dim myarray(3) As String  
  
    myArray.SetValue("zoo", 0)  
    myArray.SetValue("brief", 1)  
    myArray.SetValue("area", 2)  
    myArray.SetValue("trim", 3)  
  
    Dim x As String  
    Dim show As String  
  
    x = chr(13) & chr(10)  
  
    Dim i As Integer  
  
    For i = 0 To 3  
        show = show & myarray(i) & x  
    Next
```

```
textbox1.text = show & x & x & "SORTED:" & x

Array.Sort(myArray)

show = ""
For i = 0 To 3
    show = show & myarray(i) & x
Next

textbox1.text = textbox1.text & show

Dim anIndex As Integer

anIndex = Array.BinarySearch(myArray, "trim")
show = CStr(anIndex)
msgbox("The word trim was found at the " & show & " index
        within the array")

End Sub
```

Of course, like most objects in VB .NET, the `Array` object has many members. In addition to the common properties and methods that most objects have (such as `ToString`), several members are unique to the array class (`reverse`, `GetUpperBound`, and so on). To see all the capabilities of the array class, search for “Array members” in the VB Help feature. Here is a list of all the members of the array class: `IsFixedSize`, `IsReadOnly`, `IsSynchronized`, `Length`, `LongLength`, `Rank` (the number of dimensions), `SyncRoot`, `BinarySearch`, `Clear`, `Clone`, `Copy`, `CopyTo`, `CreateInstance`, `Equals`, `GetEnumerator`, `GetHashCode`, `GetLength`, `GetLongLength`, `GetLowerBound`, `GetType`, `GetUpperBound`, `GetValue`, `IndexOf`, `Initialize`, `LastIndexOf`, `Reverse`, `SetValue`, `Sort`, and `ToString`.

Auto List Members

The Auto List Members feature, which shows you a list of all the methods and properties of an object, has become essential in VB .NET.

If you do not see certain events or other members automatically listed — such as when you’re creating a new `UserControl` — go to `Tools` ⇨ `Options` ⇨ `Text Editor` ⇨ `Basic` ⇨ `General` and deselect `Hide Advanced Members`.

AutoRedraw Is Gone

In previous versions of VB, you could force drawn (or printed) content on a form to persist (to remain visible even after the form was minimized and restored or temporarily covered by another window). You set the `AutoRedraw` property to `True`. This property is unavailable in VB .NET. Alternatively, you could put code in the `Paint` event that would redraw your content each time the background was repainted.

Here's one way to make drawn content persist in VB .NET. Create a sub like this that overrides (overrules) the existing `OnPaint` method:

```
Protected Overrides Sub OnPaint(ByVal e As
    System.Windows.Forms.PaintEventArgs)

    With e.Graphics
        .DrawString("Name", Me.Font, Brushes.Black, 25, 25)
        .DrawString("Address", Me.Font, Brushes.Black, 25, 50)
        .DrawString("State", Me.Font, Brushes.Black, 25, 75)
    End With

End Sub
```

ByVal Becomes the Default

There are two ways to pass parameters to a function. Traditionally, VB defaulted to the `ByRef` style unless you specifically used the `ByVal` command.

Here's an example of how both `ByVal` and `ByRef` work. By default, in traditional VB, any of the variables passed to a function (or sub) can be changed by the function.

However, you can prevent the function from changing a variable if you use the `ByVal` command to protect that variable. When you use `ByVal`, the passed variable can still be used for information (can be read by the function) and even changed temporarily while within the function. But when you return to the place in your program from which you called the function, the value of the variable passed `ByVal` will not have been changed.

In the following example, `x` will not be changed, no matter what changes might happen to it while it's inside the function; `y`, however, can be permanently changed by the function:

```
Public X, Y

Private Sub Form_Load()
    X = 12
    Y = 12
    Newcost X, Y
    MsgBox "X = " & X & " Y = " & Y
End Sub

Function Newcost(ByVal X, Y)
    X = X + 1
    Y = Y + 1
End Function
```

This results in `x = 12` and `y = 13`.

We defined `X` and `Y` as `Public` — making them form-wide in scope. Both of these variables can therefore be changed within any procedure unless passed by `ByVal`. You sent both `x` and `y` to your function, but `x` is protected with the `ByVal` command. When the function adds 1 to each variable, there is a permanent change to `y`, but `x` remains unaffected by activities within the function because you froze it with the `ByVal` command.

VB .NET

The default has been changed to `ByVal`. Therefore, by default, if a procedure modifies a passed parameter, the change will not be visible to the caller (the procedure that passed the parameter).

Some parameters still default to ByVal

Not all parameters default to `ByVal` (this kind of inconsistency makes life interesting for us programmers): References to classes, interfaces, arrays, and string variables all still default to `ByRef`.

The fact that `ByVal` is now the default might require considerable rewriting to older VB code — depending on how much the programmer relied on the use of public variables and the assumption that all variables were passed by default using the `ByRef` style.

ParamArrays

Any procedure containing a `ParamArray` argument used to be able to accept any number of arguments, and it could also modify the value of any of those arguments. The calling routine used to be able to see any modifications. Now, in VB .NET, you can still use `ParamArrays`, but any modifications to the values will not be seen by the caller. This new approach is in keeping with the default to the `ByVal` style for most parameters. You cannot change this behavior by employing the `ByRef` or `ByVal` keywords. For information on how to use several different argument lists with a single procedure, see “Overloaded Functions, Properties, and Methods (Overloading)” in this appendix.

Caption Changes to Text

Versions prior to VB .NET used a `Caption` property for forms and various controls:

```
Form1.Caption = "THIS"  
Label1.Caption = "That"
```

But now there is a `Text` property for all forms and controls:

```
Me.Text = "THIS"  
Label1.Text = "That"
```

Case-Sensitivity

Before VB .NET

VB 6 and earlier was not case-sensitive. Variable names and methods or properties could be capitalized any way you wanted:

```
Dim ThisVar as String
```

You could later reference it as `Thisvar`, `thisvar`, `THISVAR`, or any other combination, and it would still be recognized by VB as the same variable.

VB .NET

If you want your objects to be usable by other languages in the Visual Studio group (namely C and its offspring, such as C++, Java, and C#), you must take care to capitalize your objects' member names (properties, methods, or events) using precisely the same capitalization. In other words, you must be case-sensitive. In the C world, `ThisVar` and `Thisvar` represent two entirely different entities.

CheckBox.Value Becomes CheckBox.Checked

The headline says it all: What used to be the `Value` property, indicating whether a `CheckBox` was checked, has become in VB .NET the `Checked` property. `Circle`, `Line`, `Shape`, `PSet`, and `Point` all changed.

Before VB .NET

These various graphics controls and commands have all been removed from the language.

VB .NET

The `System.Drawing.Graphics` class includes various methods that replace the older controls and commands. In this case, what you give up in convenience, you gain in flexibility.

Here's an example that illustrates how to use the new VB .NET graphics features to draw a square outline (which uses a pen), and then to fill that square (which uses a brush):

```
Private Sub Button1_Click_1(ByVal sender As System.Object,  
    ByVal e As System.EventArgs) Handles  
    Button1.Click  
    'create graphics object  
    Dim grObj As System.Drawing.Graphics  
    grObj = Me.CreateGraphics  
  
    'create a red pen  
    Dim penRed As New System.Drawing.Pen  
        (System.Drawing.Color.PaleVioletRed)
```

```
'make it fairly wide
penRed.Width = 9

'draw a square
grObj.DrawRectangle(penRed, 12, 12, 45, 45)

'create a brush (to fill the square)
Dim brushBlue As System.Drawing.Brush
brushBlue = New SolidBrush(Color.DarkBlue)

grObj.FillRectangle(brushBlue, 12, 12, 45, 45)

End Sub
```

Note that there are many other methods you can use with a graphics object besides the `DrawRectangle` and `FillRectangle` methods illustrated in this example. Use `DrawEllipse` to draw circles and other round objects, and also try `DrawLine`, `DrawPolygon`, and others.

The four arguments at the end of this line of code describe the horizontal (12) and vertical (15) point on the form where the upper-left corner of your rectangle is located, and the width (36) and height (45) of the rectangle:

```
grObj.FillRectangle(brushBlue, 12, 15, 36, 45)
```



The `PSet` and `Point` commands, which allowed you to manipulate individual pixels in earlier versions of VB, can be simulated by using the `SetPixel` or `GetPixel` methods of the `System.Drawing.Bitmap` class.



You can create a solid graphic (a square, circle, and so on) by simply using a brush to fill the area, without first using a pen to draw an outline.

Circle, Line, Shape, PSet, and Point All Changed

The familiar VB .NET graphics commands `Circle`, `Line`, `Shape`, `PSet`, and `Point` are all now removed from the language. Instead, you draw graphics by using a graphics object and its various methods. To outline a shape, you use a pen, and to fill a shape with color or texture, you use a brush. This is all very new to VB programmers, but it is more flexible and powerful than the old techniques. See “Drawing Directly on a Control” in this appendix.

Clipboard Becomes a Clipboard Object

Before VB .NET

To retrieve text contents from the Windows Clipboard, you used this code in VB Version 6 and earlier:

```
Text1.Text = Clipboard.GetText
```

VB .NET

You can bring text in from the clipboard using this code:

```
Dim txtdata As IDataObject = Clipboard.GetDataObject()  
  
' Check to see if the Clipboard holds text  
If (txtdata.GetDataPresent(DataFormats.Text)) Then  
    TextBox1.Text = txtdata.GetData(DataFormats.Text).ToString()  
End If
```

To export or save the contents of a TextBox to the clipboard, use this code:

```
Clipboard.SetDataObject(TextBox1.Text)
```

Closing Event

VB .NET includes a new Closing event for a form. In this event, you can trap — and abort — the user's attempt to close the form (by clicking the x in the top right or by pressing Alt+F4).

Here's an example that traps and aborts closing:

```
Private Sub Form1_Closing(ByVal sender As Object, ByVal e  
    As System.ComponentModel.CancelEventArgs) Handles  
    MyBase.Closing  
  
    Dim n As Integer  
    n = MsgBox("Are you sure you want to close this form?",  
        MsgBoxStyle.YesNo)
```

```
If n = MsgBoxResult.No Then
    e.Cancel = True
    MsgBox("This form will remain open.")
End If

End Sub
```

To see all the events available to the Form object, drop down the list at the top left of the code window. Select (Base Class Events), or if you are using VB .NET 2003, select (Form1 Events) and then drop the list at the top right of the code window to see all the events.

Compile

See “Deployment.”

Constants Drop VB Prefix

Built-in language constants such as VB 6 `vbGreen` (and a whole slew of other color constants), `vbKeyLeft` (representing the left-arrow key, plus a whole slew of other key constants), and all other built-in sets of constants no longer use that `vb` prefix in VB .NET.

For example, in VB .NET what was `vbGreen` is now simply `Green` and `vbKeyLeft` is `KeyLeft`.

If this sounds to you like a simplification, don't jump to conclusions. Here's how it looks in VB .NET:

```
Private Sub Button1_Click_1(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button1.Click

    form1.BackColor = color.Green

End Sub
```

As you can see, you must qualify the constant by using the term `color` (a namespace). You must know the class (the library) to which the constant belongs in order for the constant to actually work in your programming. Things can get a bit complex in VB .NET. For example, in VB 6, you tested which key was pressed when the user entered text into a Textbox, like this:

```
Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)

    If KeyCode = vbKeyC Then
        MsgBox ("They entered C")
    End If

End Sub
```

However, VB .NET doesn't supply a `KeyCode` to the `TextBox` `KeyDown` event. Instead, it looks like this:

```
Private Sub TextBox1_KeyDown(ByVal sender As Object, ByVal e As System.Windows.Forms.KeyEventArgs) Handles TextBox1.KeyDown

    If e.KeyCode = keys.C Then

        MsgBox("They entered C")

    End If

End Sub
```

So you must look at the `KeyEventArgs` variable, and you must look at its `KeyCode` property. However, at the current time, the key constants aren't working the same way as the color constants (simply dropping the `vb` prefix). Rather than `KeyLeft`, you must use `keys.left`.

Similarly, the VB .NET control characters (such as `CrLf`, which is used in a `MsgBox` to force the message text to appear down one line), requires the `ControlChars` namespace:

```
Dim s As String = "This line."
s += ControlChars.CrLf + "Next line."
Debug.Write(s)
```

See "Imports" for more information on this topic.

Constructors Are Now "Parametized"

When you instantiate (create) a new structure or object in VB .NET, you, the programmer, sometimes have the option of assigning a value (or sometimes multiple values) to that new object.

Here's a similar technique that you might be familiar with. In VB .NET, you can simultaneously declare a variable or array and assign a value to it:

```
Dim s As String = "Alabama"
```

However, there is a related technique — new in VB .NET — called parameterized constructors. This technique is very important to object-oriented programming. In previous versions of VB, the Initialize event could not take parameters. Therefore, you could not specify the state of an object during its initialization. (See “Initializers” in this appendix.)

Now switch viewpoints. Programmers sometimes write code that will be used by other programmers. That’s the case when you write a class — it will later be used (by a programmer or by you) to generate objects that will be useful in the application you are designing. It’s often very important to be able to force the person who uses your classes to provide initial information — to provide, in other words, parameters.

Think of it this way: When you write a function, you often force the user of that function to provide an initial parameter (or several):

```
Function AddNumbers(ByVal numone As Integer, ByVal numtwo As
    Integer) As Integer

Return numone + numtwo

End Function
```

It would make no sense to permit a programmer to use this `AddNumbers` function in his or her code without providing the two parameters that it needs to do its job. Likewise, when you write a class, it is often necessary to require that a user of your class provide parameters when that class instantiates an object. By require, I mean that VB .NET will display an error message if the programmer attempts to generate an object without the required parameters.

To repeat: When you define a class in your source code, you might want to force any programmer who later uses your class to pass a value or values. A constructor is a method of a class that is automatically executed when that class is instantiated. The purpose of a constructor is to initialize the object during its construction. Here’s a class that requires any programmer who uses it to pass it an ID number as a parameter. A method named `New` is always a constructor in a class:

```
Public Class Codecs

Private mID As String 'property named mID

Sub New(ByVal initValue As String) 'make them pass a string
    MyBase.New()
    mID = initValue ' store the string as the mID
End Sub

Public ReadOnly Property TheID()
    Get
        TheID = mID 'let them read the value
    End Get
End Class
```

```
End Get
End Property

End Class
```

In this example, the `New` method (the constructor) requires that a string be passed to this object when the object is initialized. If the programmer tries to instantiate a `Codecs` object without passing that string, VB .NET displays an error message and refuse to instantiate. Here's how you would correctly instantiate this class:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    Dim c As Codecs
    c = New Codecs("DaraLeen112")
    MsgBox("The programmer passed this value: " + c.TheID)

End Sub
```

When you press F5 and run this, the `MsgBox` shows you that the value "DaraLeen112" was indeed passed to the constructor and stored in the `TheID` property.

Permitting multiple parameters

You can also employ more than one constructor if you wish. That allows the user to choose between more than one kind of parameter list. (VB .NET differentiates between these parameter lists by noting their different variable types or noting that they contain a different number of parameters.)

VB .NET executes the constructor (the method named `New` in your class) that matches whatever parameters the programmer passes. (This is quite similar to overloading, which you can read about elsewhere in this appendix.)

For example, your class could have these two different constructors:

```
Sub New()
    MyBase.New()
    mID = Now.ToString ' generate an ID if they don't pass one
End Sub

Sub New(ByVal initValue As String) 'let them pass a string
    MyBase.New()
    mID = initValue ' use the string as the ID
End Sub
```

Now the programmer has a choice. He or she can instantiate but pass no argument:

```
c = New Codecs()
```

In this case, the `New()` procedure here with no parameters would then be executed, and the other `New(ByVal initValue As String)` would be ignored. (In this example, the first `New()` constructor generates an ID automatically using the `Now` command.)

Or, if the programmer does pass a parameter, the second constructor, `New(ByVal initValue As String)`, is executed instead.

In fact, your class can have as many constructors as you want, as long as each one has a unique argument list. Here are two additional constructors for this example (shown in boldface):

```
Public Class Codecs

    Private mID As String 'ID property value

    Sub New()
        MyBase.New()
        mID = Now.ToString ' generate an ID if they don't pass one
    End Sub

    Sub New(ByVal initValue As String) 'let them pass a string
        MyBase.New()
        mID = initValue ' use the string as the ID
    End Sub

    Sub New(ByVal initValue1 As String, ByVal initValue2 As
String) 'let them pass two strings
        MyBase.New()
        mID = initValue1 + initValue2 ' combine them to form the
ID
    End Sub

    Sub New(ByVal initValue As Integer) 'let them pass an integer
MyBase.New()
        mID = initValue.ToString ' convert their integer to a
string
    End Sub

    Public ReadOnly Property TheID()
        Get
            TheID = mID 'let them read the value
        End Get
    End Property

End Class
```

This concept is discussed further in the section titled “Overloaded Functions, Properties, and Methods (Overloading)” in this appendix.

ContainsFocus Property

A new property tells you whether or not this control (or a child control on it) has the focus, meaning that the next key pressed on the keyboard will be sent to this control. If you want to know whether the control has focus regardless of whether or not any of its child controls have the focus, use the `Focused` property instead.

ContextMenu Property

You can add context menu controls to your form from the Toolbox. A particular context menu control can be assigned to a control by specifying the context menu's name property in the `ContextMenu` property.

Control Arrays Are Gone

Before VB .NET

When you have several controls of the same type performing similar functions, grouping them into a control array was a valuable feature, allowing you to manipulate them efficiently. Also, a control array was the only way to create a new control (such as a brand new `TextBox` or a new group of buttons) while a program is running.

Grouping controls into an array lets you manipulate their collective properties quickly. Because they're now labeled with numbers, not text names, you can use them in loops and other structures (such as `Select Case`) as a unit, easily changing the same property in each control by using a single loop.

There were several ways to create a control array, but probably the most popular was to set the `index` property of a control during design time. During run time, you can use the `Load` and `Unload` commands to instantiate new members of this array.

Each control in a control array gets its own unique index number, but they share every event in common. In other words, one `Click` event, for example, would be shared by the entire array of controls. An `index` parameter specifies which particular control was clicked. So you would write a `Select Case` structure like the following within the shared event to determine which of the controls was clicked and to respond appropriately:

```
Sub Buttons_Click (Index as Integer)

    Select Case Index
    Case 1
        MsgBox ("HI, you clicked the OK Button!")
    Case 2
        MsgBox ("Click the Other Button. The one that says OK!")
    End Select
End Sub
```

There is a way to simulate this all-in-one event that handles all members of a control array in VB .NET. I describe this simulation in the upcoming section titled “Multiple handles.”

VB .NET

Control arrays have been removed from the language. However, in VB .NET, you can still do what control arrays did. You can instantiate controls during run time and also manipulate them as a group.

To accomplish what control arrays used to do, you must now instantiate controls (as objects) during run time and then let them share events (even various different types of controls can share an event). Which control (or controls) is being handled by an event is specified in the line that declares the event (following the `Handles` command, as you can see in the next example). Instead of using index numbers to determine what you want a control to do (when it triggers an event), as was the case with control arrays, you must now check an object reference. (Each instantiated control should be given its own, unique name; programmer-instantiated controls do not automatically get a `Name` property specified for them by default.) You are also responsible for creating events for run time-generated controls. The `Name` property can now be changed during runtime.

Here’s an example showing how you can add new controls to a form while a program is running.

Adding a button to a form

Assume that the user clicked a button asking to search for some information. You then create and display a `TextBox` for the user to enter the search criteria, and you also put a label above it describing the `TextBox`’s purpose:

```
Public Class Form1

    Inherits System.Windows.Forms.Form

    Dim WithEvents btnSearch As New Button()

    Private Sub Button1_Click(ByVal sender As Object, ByVal e As
        System.EventArgs) Handles Button1.Click

        Dim textBox1 As New TextBox()
        Dim label1 As New Label()

        ' specify some properties:
        label1.Text = "Enter your search term here..."
        label1.Location = New Point(50, 55) 'left/top
        label1.Size = New Size(125, 20) ' width/height
        label1.AutoSize = True
        label1.Name = "label1"

        textBox1.Text = ""
        textBox1.Location = New Point(50, 70)
        textBox1.Size = New Size(125, 20)
        textBox1.Name = "TextBox1"

        btnSearch.Text = "Start Searching"
        btnSearch.Location = New Point(50, 95)
        btnSearch.Size = New Size(125, 20)
        btnSearch.Name = "btnSearch"

        ' Add them to the form's controls collection.
        Controls.Add(textBox1)
        Controls.Add(label1)
        Controls.Add(btnSearch)

        'display all the current controls

        Dim i As Integer, n As Integer
        Dim s As String
        n = Controls.Count

        For i = 0 To n - 1
            s = Controls(i).Name
            Debug.Write(s)
            Debug.WriteLine("")
        Next

        End Sub

    Private Sub btnSearch_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles
        btnSearch.Click
```

```
MsgBox("clicked")  
  
End Sub  
  
End Class
```

When adding new controls at design time, you want to at least specify their name, size, and position on the form — especially their `Name` property. Then, use the `Add` method to include these new controls in the form's controls collection.



VB .NET programmers will expect VB .NET to assign names to dynamically added controls. However, be warned that VB .NET does not automatically assign names to new controls added at design time. Therefore, the `Name` property remains blank unless you specifically define it, as you did in the preceding example (`textBox1.Name = "TextBox1"`).

Here's how to go through the current set of controls on a form and change them all at once. The following example turns them all red:

```
n = Controls.Count  
  
For i = 0 To n - 1  
    Controls(i).BackColor = Color.Red  
Next
```

Of course, a control without any events is often useless. To add events to run time–created controls, you must add two separate pieces of code. First, up at the top (outside any procedure, because this declaration cannot be local), you must define the control as having events by using the `WithEvents` command, like this:

```
Dim WithEvents btnSearch As New Button()
```

Then, in the location where you want to provide the code that responds to an event, type the following:

```
Private Sub btnSearch_Click(ByVal sender As System.Object,  
    ByVal e As System.EventArgs) Handles  
    btnSearch.Click  
  
    MsgBox("clicked")  
  
End Sub
```

This event code is indistinguishable from any other “normal” event in the code. Notice that unlike the all-in-one event shared by all the controls in the entire control array in VB 6, VB .NET expects you to give each newly created control its own name and then use that name to define an event that uses “Handles” `Name.Event`, as illustrated by the `Click` event for `btnSearch` in the previous example.

Multiple handles

If you're wildly in favor of the all-in-one event, you can do some curious things in VB .NET that permit it. Here's an example that handles the `Click` events for three different controls by declaring their names following the `Handles` command:

```
Private Sub cmd_Click(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles cmdOK.Click,
    cmdApply.Click, cmdCancel.Click

    Select Case sender.Name
        Case "cmdOK"
            'Insert Code Here to deal with their clicking
            'the OK button
        Case "cmdApply"
            'Insert Code Here for Apply button clicking
        Case "cmdCancel"
            'Insert Code Here for Cancel button clicks
    End Select

End Sub
```

In addition to the multiple objects listed after `Handles`, notice another interesting thing in this code. There's finally a use for the sender parameter that appears in every event within VB .NET. It is used in combination with the `.Name` property to identify which button was, in fact, clicked. This event handles three `Click` events. `Sender` tells you which control raised (triggered) this event at any given time. In most VB .NET code, the `Sender` object is the same as both the name of the event and the name following the `Handles`:

```
Private Sub Button1_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button1.Click

    MsgBox(sender.Name)

End Sub
```



VB .NET creates the event in the code window for you, if you wish. Your `btnSearch` doesn't show up in the Design window, so you cannot double-click it there to force VB .NET to create a `Click` event for it. However, you can use the drop-down lists. After you have declared a control `WithEvents` (`Dim WithEvents btnSearch As New Button()`), drop the list in the top left of the code window, and locate `btnSearch`. Click it to select it. Then drop the list in the top right and double-click the event that you want VB .NET to create for you in the code window.



Each Form has a collection that includes all the controls on that form. You access the collection, as illustrated previously, by using `Me.Controls` or simply `Controls`. The collection can be added to, as shown in the previous example, or can be subtracted from:

```
Me.Controls.Remove(Button1)
```

Note, too, that the `Me.Controls` collection also has several other methods: `Clear`, `Equals`, `GetChildIndex`, `GetEnumerator`, `GetHashCode`, `GetType`, `SetChildIndex`, `ShouldPersistAll`, and `ToString`. There are also three properties available to `Me.Controls`: `Count`, `Item`, and `IsReadOnly`.

Controls Property

This new property represents a collection of any child controls within the current control.

Converting Data Types

VB .NET offers four ways to change one data type into another. The `.ToString` method is designed to convert any numeric data type into a text string.

The second way to convert data is to use the traditional VB functions: `CStr()`, `CBool()`, `CByte()`, `CChar()`, `CShort()`, `CInt()`, `CLng()`, `CDate()`, `Cdbl()`, `CSng()`, `CDec()`, and `CObj()`. Here is an example:

```
Dim s As String
Dim i As Integer = 1551
s = CStr(i)
MessageBox.Show(s)
```

The third way is to use the `Convert` method, like this:

```
Dim s As String
Dim i As Integer = 1551
s = Convert.ToString(i)
MessageBox.Show(s)
```

The fourth way uses the `CType` function, with this syntax:

```
Dim s As String
Dim i As Integer = 1551
s = CType(i, String)
MessageBox.Show(s)
```

See “Option Strict and Option Explicit” for more information.

CStr Changes to ToString

Many classic VB built-in functions are still available in VB .NET. Often, there is no difference between the classic function's behavior and the new .NET version. For example, if you want to convert a variable to a string-type variable, you can either use the classic `CStr` or the new VB .NET `ToString` method:

```
Dim n As Integer
n = 12

MsgBox(CStr(n))
MsgBox(n.ToString)
```

Note that, by default, VB .NET doesn't insist that you make these conversions; it will display 12 if you merely use this code:

```
MsgBox(n)
```

However, if you want to enforce strict conversion rules, making explicit conversion from one data type to another a requirement, add this line at the very top of your code window:

```
Option Strict On
```

For more information about `Option Strict`, see “`Option Strict` and `Option Explicit`.”

Yet another way to convert variables from one type to another is to use the `Convert` method:

```
Dim s As String
Dim i As Integer = 1551
s = Convert.ToString(i)
MessageBox.Show(s)
```

Also see “`Converting Data Types`.”

Currency Data Type Is No More

Before VB .NET

The `Currency` data type (a scaled integer) used the symbol `@` and could hold the following range of numbers:

```
-922,337,203,685,477.5808 to 922,337,203,685,477.5807
```

It specialized in large numbers that needed only four decimal places of precision.

VB .NET

The `Currency` data type has been removed from the language. You are advised to rely on the `Decimal` data type in its stead. See “Data Types” in this appendix for more information.

Cursor Property

The new `Cursor` property is what used to be called the `MouseIcon` property, and it determines what the mouse pointer looks like when it’s on top of the `TextBox` (if you want to change it from the default pointer). I advise against changing this property — unless you’re sure you will not confuse the user.

Data Types

There is no `Variant` data type in VB .NET. There are some other changes to the traditional VB data types. A new `Char` type, which is an unsigned 16-bit type, is used to store Unicode characters. The `Decimal` type is a 96-bit signed integer scaled by a variable power of 10. No `Currency` type is included anymore (use the `Decimal` type instead).

The VB .NET `Long` data type is now a 64-bit integer. The VB .NET `Short` type is a 16-bit integer. And in between them is the new VB .NET `Integer` type, which is a 32-bit value. If you are working with programming in which these distinctions will matter, memorize these differences from the way that integers were handled in traditional VB programming.

Use the `Integer` type when possible because it executes the fastest of the comparable numeric types in VB .NET.

Here are all the VB .NET data types:

<i>Traditional VB Type</i>	<i>New .NET Type</i>	<i>Memory Size</i>	<i>Range</i>
Boolean	<code>System.Boolean</code>	4 bytes	True or False
Char	<code>System.Char</code>	2 bytes	0–65535 (unsigned)
Byte	<code>System.Byte</code>	1 byte	0–255 (unsigned)

(continued)

(continued)

Traditional VB Type	New .NET Type	Memory Size	Range
Object	System.Object	4 bytes	Any Type
Date	System.DateTime	8 bytes	01-Jan-0001 to 31-Dec-9999
Double	System.Double	8 bytes	+/-1.797E308
Decimal	System.Decimal	12 bytes	28 digits
Integer	System.Int16	2 bytes	-32,768 to 32,767
	System.Int32	4 bytes	+/-2.147E9
Long	System.Int64	8 bytes	+/-9.223E18
Single	System.Single	4 bytes	+/-3.402E38
String	System.String	CharacterCount * 2 (plus 10 bytes)	2 billion characters

Also see “Converting Data Types.”

Date and Time: New Techniques

Before VB .NET

The `Date` function used to give you the current date (for example: 11/29/00).
 The `Time` function used to give you the current time.

VB .NET

Now you must use the `Today` and `TimeOfDay` functions instead. Also note that the old `DATE$` and `TIME$` functions have been eliminated.

In Visual Basic 6.0 and previous versions, a date/time was stored in a double (double-precision floating point) format (4 bytes). In VB .NET, the date/time information uses the .NET framework `DateTime` data type (stored in 8 bytes). There is no implicit conversion between the `Date` and `Double` data types in VB .NET. To convert between the VB 6 `Date` data type and the VB .NET `Double` data type, you must use the `ToDouble` and `FromODate` methods of the `DateTime` class in the `System` namespace.

Here's an example that uses the `TimeSpan` object to calculate how much time elapsed between two `DateTime` objects:

```
Dim StartTime, EndTime As DateTime
Dim Span As TimeSpan

    StartTime = "9:24 AM"
    EndTime = "10:14 AM"

    Span = New TimeSpan(EndTime.Ticks - StartTime.Ticks)
    MsgBox(Span.ToString)
```

Notice the `Ticks` unit of time. It represents a 100-nanosecond interval.

Here's another example illustrating the `AddHours` and `AddMinutes` methods, how to get the current time (`Now`), and a couple other methods:

```
Dim hr As Integer = 2
Dim mn As Integer = 13

Dim StartTime As New DateTime(DateTime.Now.Ticks)
Dim EndTime As New DateTime(StartTime.AddHours(hr).Ticks)

EndTime = EndTime.AddMinutes(mn)

Dim Difference = New TimeSpan(EndTime.Ticks -
    StartTime.Ticks)

Debug.WriteLine("Start Time is: " +
    StartTime.ToString("hh:mm"))
Debug.WriteLine("Ending Time is: " +
    EndTime.ToString("hh:mm"))
Debug.WriteLine("Number of hours elapsed is: " +
    Difference.Hours.ToString)
Debug.WriteLine("Number of minutes elapsed is: " +
    Difference.Minutes.ToString)
```

The following sections provide some additional examples that illustrate how to manipulate date and time.

Adding time

Here's an example of using the `AddDays` method:

```
Dim ti As Date = TimeOfDay 'the current time
Dim da As Date = Today 'the current date
Dim dati As Date = Now 'the current date and time

    da = da.AddDays(12) ' add 12 days
    Debug.WriteLine("12 days from now is:" & da)
```

Similarly, you can use `AddMinutes`, `AddHours`, `AddSeconds`, `AddMilliseconds`, `AddMonths`, `AddYears`, and so on.

Using the old-style double `DateTime` data type

There is an OA conversion method for Currency data types and for Date data types. (OA stands for Ole Automation, a legacy technology that still keeps popping up.) Here's an example showing how to translate to and from the old double-precision date format:

```
Dim dati As Date = Now 'the current date and time
Dim da as Date
n = dati.ToOADate ' translate into double-precision format
n = n + 21 ' add three weeks (the integer part is the days)
da = Date.FromOADate(n) ' translate the OA style into .NET
    style
Debug.WriteLine(da)
```



Use `Now`, not `Today`, for these OA-style data types.

Finding days in month

2004 is a leap year. Here's one way to prove it:

```
Debug.WriteLine("In the year 2004, February has " &
    Date.DaysInMonth(2004, 2).ToString & " days.")
Debug.WriteLine("In the year 2005, February has " &
    Date.DaysInMonth(2005, 2).ToString & " days.")
```

Debug.Print Changes

If you're one of those programmers who likes to use `Debug.Print` while testing your project, note that that feature has been changed in VB .NET to `Debug.Write`. If you want to force a linefeed, use `Debug.WriteLine`.

Also, instead of displaying its results in the Immediate window (as in previous versions of VB), `Debug.Write` now writes to the Output window.

Default Control Names Are Now Longer

Before VB .NET

When you added a `TextBox` to a form, it was given a default name `Text1`. Similarly, when you added a `ListBox`, it was named `List1`. It doesn't work this way anymore.

VB .NET

For reasons I don't understand, VB .NET now provides longer default names: `TextBox1`, `ListBox1`, and so on.

Default Properties Are No More

You can no longer use code like this, because default properties (such as `Text1.Text`) are no longer permitted with controls:

```
Text1 = "This phrase"
```

Instead, in VB .NET, you must specify the property:

```
Text1.Text = "This phrase"
```

See the sections “Object Instantiation” and “Property Definitions Collapse (Property Let, Get, and Set Are Gone)” in this appendix for more information.

Deployment

You have ways to give others your WinForm-based and WebForm-based applications. See VB .NET Help for ways to deploy your solutions. However, if you are wondering how to write a little utility program in VB .NET and run it in your local computer, you'll doubtless notice that the usual `File→Make .EXE` or `File→Compile .EXE` option is missing in VB .NET.

Don't despair, though. Whenever you press F5 to test your VB .NET WinForm-style project, it is compiled into an `.exe` program with the same name as your project. You can find this `.exe` file in: `C:\Documents and Settings\YourIdentityName\MyDocuments\Visual Studio Projects\YourProjectsName\Bin`.

Dim and ReDim: New Rules

Before VB .NET

The `Dim` command could either dimension an array or declare a variable (or a group of variables).

The default variable type was the variant, so if you wanted to use a different type, you had to explicitly declare it using `Dim` or one of the other declaring commands, such as `Public`. (Variants are not permitted in VB .NET.)

You could `Dim` a list of different variables:

```
Dim X As String, Y As Integer, Z
```

In this example, `Z` would default to `Variant` because you didn't specify a data type.

Or you could use the `DefType` commands to change the default from `Variant` to, say, `Integer`:

```
DefInt A-Z '(all variables become integer types unless  
otherwise declared).
```

The `ReDim` command could be used two ways: to resize arrays or to declare variables.

VB .NET

The `DefType` commands have been removed from the language, as has the `Variant` data type. The now-removed `DefType` commands were `DefBool`, `DefByte`, `DefCur`, `DefDate`, `DefDbl`, `DefDec`, `DefInt`, `DefLng`, `DefObj`, `DefSng`, `DefStr`, and `DefVar`.

You can mix types within a `Dim` list, but all variables are required to have an `As` clause defining their type (if you set `Option Strict On`). This next line of code is not permitted because of the `Z`:

```
Dim X As String, Y As Integer, Z
```

This is allowed:

```
Dim X As String, Y As String, z As Integer
```

Or:

```
Dim X, Y As String
```

Note that in VB 6, the `X` (because no `As` phrase defines it) would have been a `Variant` type. In VB .NET, `X` is a `String` type.

If you leave `Option Strict Off`, and you don't attach an `As` clause, the variable will default to the `Object` type:

```
Dim X
```

See “Option Strict and Option Explicit” for more details.

You must declare every variable and every array. You must use `Dim` (or another declaring command, such as `Private` or `Public`). `Option Explicit` is built into the language as the default. (`Option Explicit` requires that every variable be declared, though you can turn it off.) The default type is now `Object` instead of `Variant`, but you must declare even this “default” type.



You can no longer implicitly declare a variable merely by assigning a value to it, like this:

```
X = 12
```

However, VB .NET does allow you to assign a value in the same statement that declares the variable:

```
Dim X As Integer = 12
```

Also, if you are using the latest version, VB .NET 2003, you can even declare a loop counter variable within the line that starts the loop, like this:

```
For i As Integer = 0 To 10
```

instead of the traditional style:

```
Dim i as Integer  
For i = 0 To 10
```

You can no longer use `ReDim` in place of the `Dim` command. Arrays must be first declared using `Dim` (or similar); `ReDim` cannot create an array. You can use `ReDim` only to redimension (resize) an existing array originally declared with the `Dim` (or similar) command. (See “ArrayList (Replacing the Array Command)” for a superior way to let VB .NET handle arrays that must dynamically change size. `ArrayList` also does other neat tricks.)

You can declare arrays in a similar way. Here’s a two-dimensional array declaration:

```
Dim TwoArray(3, 4) As Integer
```

Use empty parentheses to declare a dynamic array:

```
Dim DynArray() As Integer
```



You are allowed to declare the same variable name in more than one location. This can cause problems that are not flagged by any error messages. Here’s an example: Assume that you originally declared `n` within the `Form_Load` event (so that it’s local to that event only). However, you decide you need to make it `Public` so that other procedures can use its contents as well. Therefore, you type in a `Public` declaration at the top, outside of any event

or procedure. But you forget to erase the local declaration. Unfortunately, you get hard-to-track-down bugs because the local declaration will remain in effect and `n` will lose its contents outside the `Form_Load` event. Here's an example:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Public n As Integer '(holds the number of files being
        renamed)

    Private Sub Form1_Load(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles MyBase.Load

        Dim myfile As String
        Dim mydir As String
        Dim n, i As Integer

    End Sub
```

Strongly typed

VB .NET is strongly typed. Several changes to the language support this new approach. For one thing, the `As Any` command has been deleted from the language. You can no longer declare a function `As Any` (meaning that it can return any kind of data type). For additional details and an alternative approach, see “Overloaded Functions, Properties, and Methods (Overloading),” elsewhere in this appendix.

Declaring arrays

You declare an array in a way similar to the way you declare a variable. To create an array that holds 11 values, ranging from `MyArray(0)` to `MyArray(10)`, type the following:

```
Dim MyArray(10)
```

You can simultaneously declare and initialize (provide values to) an array (see “Initializers” in this appendix). You use the braces punctuation mark, which has never before been used in Visual Basic. Here's how to initialize a string array with two elements, `MyArray(0)` and `MyArray(1)`, which contain “Billy” and “Bob”:

```
Dim MyArray() As String = {"Billy", "Bob"}
MsgBox(MyArray(0) + MyArray(1))
```

Notice that you are not permitted to specify the size of an array when you initialize it with values, as illustrated in the previous example: `MyArray()`. This array has two elements: `(0)` and `(1)`.

Multidimensional arrays are declared as you would expect. The following array has two dimensions, each with elements ranging from 0 to 3:

```
Dim MyArray(3,3)
```

Declaring with symbols

You can still use symbols when declaring some data types in VB .NET. For example, the following declares `N` as an Integer variable:

```
Dim N%
```

The following code is equivalent:

```
Dim N as Integer
```

Here are the symbols you can use in VB .NET:

<i>Symbol</i>	<i>VB 6</i>	<i>VB .NET</i>
%	16-bit Integer	32-bit Integer
&	32-bit Long Integer	64-bit Long Integer
!	Single Floating Point	Single Floating Point
#	Double Floating Point	Double Floating Point
@		Decimal
\$	String	String

Dim As New

Before VB .NET

Here's an example of how `Dim` was used before VB .NET:

```
Dim MyObj as New MyClass
```

You could always access the members of this object because VB handled the references automatically.

VB .NET

An instance of an object is created only when the procedure in which it is declared `As New` is called. Other references to this object's properties or methods will not instantiate it automatically, as before. So you must now explicitly test whether or not the object exists:

```
If MyObj Is Nothing Then
```

Use the old VB 6 syntax, and VB .NET gives you the following error message:

```
This keyword does not name a class.
```

Dim Gets Scoped

Before VB .NET

The `Dim` command worked the same no matter where you put it inside a given procedure. These two uses of `Dim` behaved the same:

```
Private Sub Form_Load()  
  
    For I = 2 To 40  
        Dim n As Integer  
        n = n + 1  
    Next I  
End Sub  
  
Private Sub Form_Load()  
    Dim n As Integer  
  
    Do While n < 40  
        n = n + 1  
    Loop  
End Sub
```

In traditional VB, the compiler first searches through procedures like those in the example to find any `Dim` commands. Then it carries out the `Dim` commands first, before other code in the procedure. It doesn't matter where in the procedure these `Dim` commands are located. There was no scope narrower than a single procedure.

VB .NET

Now `Dim` can have a narrower scope than you ever imagined. `Dim`, when located within a code block (such as `If . . . Then`) has scope only within that code block. That's right: `Dim` inside a code block doesn't have procedure-wide scope; it works only within the block. So, if you've been in the habit of sticking your `Dim` statements here and there in your code, you'll have to clean up your act and place them at the top of procedures or outside of procedures in what used to be the General Declarations section of a form. In other words, you'll have to check each `Dim` in your existing code to ensure that it has the scope you want it to have.

Directory (Current)

Before VB .NET

To find out information about the currently executing application in VB 6, you could use the `App` object for some info and make API calls for other info.

VB .NET

In VB .NET, you can find the current directory (the one that your application is running in, for example) with this code:

```
Debug.WriteLine(System.Environment.CurrentDirectory)
```

The `System.Environment` object allows you to find out many other aspects of the environment (the currently executing process) as well:

```
Private Sub Button1_Click(ByVal sender As System.Object,  
    ByVal e As System.EventArgs) Handles  
    Button1.Click  
  
    Debug.WriteLine(Environment.CurrentDirectory)  
    Debug.WriteLine(Environment.CommandLine) 'running program  
    Debug.WriteLine(Environment.MachineName)  
    Debug.WriteLine(Environment.OSVersion)  
    Debug.WriteLine(Environment.SystemDirectory)  
    Debug.WriteLine(Environment.UserDomainName)  
    Debug.WriteLine(Environment.UserName)  
    Debug.WriteLine(Environment.Version)  
  
End Sub
```

This results in:

```
E:\Documents and Settings\Richard Mansfield.DELL\My
  Documents\Visual Studio
  Projects\WindowsApplication11\bin
"E:\Documents and Settings\Richard Mansfield.DELL\My
  Documents\Visual Studio
  Projects\WindowsApplication11\bin\WindowsApplicati
  on11.exe"

DELL
Microsoft Windows NT 5.0.2195.0
E:\WINNT\System32
DELL
Richard Mansfield
1.0.2914.16
```

Directory and File: Creating and Destroying

The following code illustrates how to create or delete directories and subdirectories:

```
Imports System.io

Public Function DestroyDirectory()

Dim objDir As New DirectoryInfo("C:\TestDir")
Try
  objDir.Delete(True)
Catch
  Throw New Exception("Failed to delete")
End Try

End Function

Public Function CreateDirectory() As String

Dim objDir As New DirectoryInfo("c:\TestDir")

Try
  objDir.Create()

Catch
  Throw New Exception("Failed to create new directory")
End Try
```

```
End Function

Public Function CreateSubDirectory() As String

Dim objDir As New DirectoryInfo("c:\TestDir") 'parent
    directory
Try
    objDir.CreateSubdirectory("TestSubDir") 'name for new
        subdiretory
Catch
    Throw New Exception("Failed to create new subdirectory")
End Try
End Function
```

A similar set of file copy, move, create, and destroy methods are available using code like this:

```
Dim fa As New FileInfo("c:\textq.txt")
fa.CopyTo("c:\testq.bak")
```

Directory-, Drive-, and FileListBoxes Are Gone

The venerable Directory-, Drive-, and FileListBox controls are now missing from the Toolbox. No great loss, because the SaveFileDialog and OpenFileDialog controls do a much better job of providing a dialog box that gives users access to hard drives.

Divide by Zero Equals Infinity

Dividing by zero is not an error in VB .NET (except for integers). Try the following:

```
Dim x As Single
Dim y As Single
y = 1 / x 'divide by zero
MsgBox(y)

If [Single].IsInfinity(y) Then
    MsgBox("yes, it's infinity")
End If
```

The first MsgBox will display Infinity. Then the second MsgBox pops up and says yes, it's infinity.

Note how you can test for “positive” or “negative” or either kind of infinity using the `IsPositiveInfinity`, `IsNegativeInfinity`, or `IsInfinity` methods of the `Single` or `Double` type. Also note the odd punctuation. This is the first time in Visual Basic that brackets have been used in the language. (New, as well, are braces — used when declaring and initializing an array. See “Dim and ReDim: New Rules.”)

Dock Property

This new property specifies which side of the parent container (normally the form) this control attaches itself to.

Drawing Directly on a Control

In VB .NET, you can get pretty down and dirty and take charge of precisely how a control will look to the user. Here's how to frame a Button control with blue. First, up near the top, next to the Form's declaration, add this line (shown in boldface):

```
'Put this WithEvents line up with the Inherits line:  
  
Public Class Form1  
  
    Inherits System.Windows.Forms.Form  
  
    Friend WithEvents Button1 As System.Windows.Forms.Button
```

Then put a Button control on your form and type this into its Paint event:

```
Private Sub Button1_Paint(ByVal sender As Object, ByVal e As  
    System.Windows.Forms.PaintEventArgs) Handles  
    Button1.Paint  
  
    Dim g As Graphics = e.Graphics  
    ControlPaint.DrawBorder(g, e.ClipRectangle, Color.Blue,  
        ButtonBorderStyle.Solid)  
  
End Sub
```

Empty, Null, Missing, IsNull, IsObject, and IsMissing Are Gone

Before VB .NET

The traditional VB `Variant` data type could hold some special kinds of values: `Null` (not known), `Empty` (no value was ever assigned to this variable), and `Missing` (this variable was not sent, for example, as part of a procedure's parameters).

`Null` was sometimes used to identify fields in databases that are not available (or unknown), while the `Empty` command was used to represent something that didn't exist (as opposed to simply not being currently available).

Some programmers used the `IsMissing` command to see if an optional parameter had been passed to a procedure:

```
Sub SomeSub(Optional SomeParam As Variant)
    If IsMissing(SomeParam) Then
```

VB .NET

The VB .NET `Object` data type does not use `Missing`, `Empty`, or `Null`. There is an `IsDBNull` function that you can use with databases instead of the now-missing `IsNull` command. Similarly, an `IsReference` command replaces the `IsObject` command.

Optional parameters can still be used with procedures, but you must declare that `As Type` and you must also supply a default value for them. You cannot write code within a procedure that will tell you whether a particular parameter has been passed. For further details, see the section titled “Optional Parameter-Passing Tightened,” in this appendix.

If you need to test whether an optional parameter has been passed, you can overload a procedure. Overloading is a new technique in which a function (or indeed a method or property) can be made to behave differently based on what is passed to it. For more on overloading, see the section titled “Overloaded Functions, Properties, and Methods (Overloading)” in this appendix.

Also see “IsEmpty.”

Environment Information

See “Directory (Current).”

Error Messages (Obscure)

Some .NET error messages can be confusing. They merely offer some general abstract comment, without telling you precisely the punctuation or syntax you must use to fix the problem.

For example, you might get an error message like this:

```
Value of type system.drawing.color cannot be converted to 1-  
dimensional array of system.drawing.color.
```

This code results in that error message:

```
Dim ArrColor() As Color = Color.Gold
```

To fix the problem, enclose the color in braces:

```
Dim ArrColor() As Color = {Color.Gold}
```

Error Handling Revised

Before VB .NET

Don't panic. You can still use the familiar VB error-handling techniques (On Error...).

Thus, you don't have to revise this aspect of your older programs. But for new programs that you write in VB .NET, you might want to consider the possibility that a superior error-trapping and handling approach exists. They call it *structured error handling*, which implies that your familiar, classic VB error handling is . . . well . . . unstructured.

If you try to write something traditional, like `If Err Then`, however, you'll be informed that VB .NET doesn't permit the `ErrObject` to be treated as `Boolean` (true/false). However, where there's a will, there's a way. You can test the `Err` object's number property. So, if you want to test the `Err` object within an `If . . . Then` structure, use this VB .NET code:

```
x = CInt(textbox1.Text)

If err.Number <> 0 Then

    textbox1.Text = "You must enter a number..."

End If
```

VB .NET

Consider using the new Try...Catch...Finally structure:

```
Sub TryError()

Try

    Microsoft.VisualBasic.FileOpen(5, "A:\Test.Txt",
        OpenMode.Input)

Catch er As Exception
    MessageBox.Show(er.ToString)
Finally

End Try

End Sub
```

Code between the Try and End Try commands is watched for errors. You can use the generic Exception (which will catch any error) or merely trap a specific exception, such as the following:

```
Catch er As DivideByZeroException
```

The term *exception* is used in C-like languages (and now in VB .NET) to mean error.

I used `er`, but you can use any valid variable name for the error. Or you can leave that variable out entirely and just use Catch, like this:

```
Try

    Microsoft.VisualBasic.FileOpen(5, "A:\Test.Txt",
        OpenMode.Input)

Catch

    MessageBox.Show("problem")
Finally

End Try
```

If an error occurs during execution of the source code in the `Try` section, the following `Catch` section is then executed. You must include at least one `Catch` section, but there can be many such sections if you need them to test and figure out which particular error occurred. A series of `Catch` sections is similar to the `Case` sections in `Select...Case` structures. The `Catch` sections are tested in order, and only one `Catch` block (or none) is executed for any particular error.

You can use a `When` clause to further specify which kind of error you want to trap, like this:

```
Dim Y as Integer
Try

Y = Y / 0
Catch When y = 0
    MessageBox.Show("Y = 0")
End Try
```

Or you can specify a particular kind of exception, thereby narrowing the number of errors that will trigger this `Catch` section's execution:

```
Catch er As ArithmeticException
    MessageBox.Show("Math error.")
Catch When y = 0
    MessageBox.Show("Y = 0")
End Try
```

To see a list of the specific exceptions, use the VB .NET menu `Debug` → `Windows` → `Exceptions` and then expand the `Common Language Runtime Exceptions`. You may have to do a bit of hunting. For instance the `FileNotFoundException` is located two expansions down in the hierarchy: `Common Language Runtime` → `SystemException` → `IOException`. So you have to expand all three (click the + next to each) in order to finally find `FileNotFoundException`.

Also notice in the `Exceptions` window that you can cause the program to ignore any of the exceptions (click the `Continue` radio button in the `Exceptions` window). This is the equivalent of `On Error Resume Next` in VB 6.

Here is a list of common errors you can trap in VB .NET. The following errors are in the `System` namespace:

```
AppDomainUnloadedException
ApplicationException
ArgumentException
ArgumentNullException
ArgumentOutOfRangeException
ArithmeticException
```

```
ArrayTypeMismatchException  
BadImageFormatException  
CannotUnloadAppDomainException  
ContextMarshalException  
DivideByZeroException  
DllNotFoundException  
DuplicateWaitObjectException  
EntryPointNotFoundException  
Exception  
ExecutionEngineException  
FieldAccessException  
FormatException  
IndexOutOfRangeException  
InvalidCastException  
InvalidOperationException  
InvalidProgramException  
MemberAccessException  
MethodAccessException  
MissingFieldException  
MissingMemberException  
MissingMethodException  
MulticastNotSupportedException  
NotFiniteNumberException  
NotImplementedException  
NotSupportedException  
NullReferenceException  
ObjectDisposedException  
OutOfMemoryException  
OverflowException  
PlatformNotSupportedException  
RankException  
ServicedComponentException  
StackOverflowException  
SystemException  
TypeInitializationException  
TypeLoadException  
TypeUnloadedException  
UnauthorizedAccessException  
UnhandledExceptionEventArgs  
UnhandledExceptionHandler  
UriFormatException  
WeakReferenceException
```

The following errors are in the `SystemIO` category:

```
DirectoryNotFoundException  
EndOfStreamException  
FileNotFoundException  
InternalBufferOverflowException  
IOException  
PathTooLongException
```

You can list as many `Catch` phrases as you want and respond individually to them. You can respond by notifying the user as you did in the previous example, or merely by quietly fixing the error in your source code following the `Catch`. You can also provide a brief error message with the following:

```
e.Message
```

Or, as you did in the previous example, use the following fully qualified error message:

```
e.ToString
```

Here's the full `Try...Catch...Finally` structure's syntax:

```
Try trystatements

[Catch1 [exception [As type]] [When expression]
  catchStatements1
[Exit Try]
Catch2 [exception [As type]] [When expression]
  catchStatements2
[Exit Try]
...
Catchn [exception [As type]] [When expression]
  catchStatementsn
[Exit Try]
[Finally
  finallyStatements]
End Try
```

Recall that following the `Try` block, you list one or more `Catch` statements. A `Catch` statement can include a variable name and an `As` clause defining the type of exception or the general "all errors," `As Exception` (or `As Exception`). For example, here's how to trap all exceptions:

```
Try
  Microsoft.VisualBasic.FileOpen(5, "A:\Test.Txt",
    OpenMode.Input)

Catch e As Exception
  'Respond to any kind of error.

Finally
End Try
```

And here is how to respond to the specific `File Not Found` error:

```
Try
Microsoft.VisualBasic.FileOpen(5, "A:\Test.Txt",
    OpenMode.Input)
Catch FileNotFoundE As FileNotFoundException
    'Respond to this particular error here, perhaps a
    messagebox to alert the user.
Finally
End Try
```

An optional `Exit Try` statement causes program flow to leap out of the `Try` structure and to continue execution with whatever follows the `End Try` statement.

The `Finally` statement should contain any code that you want executed after error processing has been completed. Any code in the `Finally` section is *always* executed, no matter what happens (unlike source code following the `End Try` line, which may or may not execute, depending on how things go within the `Try` structure). Therefore, the most common use for the `Finally` section is to free up resources that were acquired within the `Try` block. For example, if you were to acquire a `Mutex` lock within your `Try` block, you would want to release that lock when you were done with it, regardless of whether the `Try` block exited with a successful completion or an exception (error). It's typical to find this kind of code within the `Finally` block:

```
objMainKey.Close()
objFileRead.Close()
objFilename.Close()
```

Use this approach when you want to close, for instance, an object reference to a key in the Registry, or to close file references that were opened during the `Try` section (“block”) of code.

Here's how source code you put within the `Finally` section differs from source code you put following the `End Try` line.

If there is an error, here is the order in which code execution takes place:

1. `Try` section
2. `Catch` section (the `Catch` section that traps this error)
3. `Finally` section

If no error occurs, here is the execution order:

1. Try section
2. Finally section
3. Source code following the End Try line



Even if a Catch section has an Exit Sub command, the Finally section nevertheless will still be executed. Finally is always executed. However, the Exit Sub does get executed just after the Finally block.

Event Handlers

See “Handles” and “Multiple handles,” under the main heading “Control Arrays Are Gone.”

Executable

See “Deployment.”

File I/O Commands Are Different

Before VB .NET

The familiar syntax is as follows:

```
Open filepath {For Mode}{options}As {#} filename {Len =  
recordlength}
```

For example:

```
Open "C:\Test.Txt" As 5
```



Some people refer to the filename as the *channel*.

VB .NET

To use the traditional VB file I/O commands, you must import the VB “Compatibility” namespace. So include this line at the top of your code window:

```
Imports Microsoft.VisualBasic
```

If you want to let your user decide which file to display or access, you want to use the handy and effective `OpenFileDialog` control found on the Toolbox.

VB .NET has a C-like “stream” (incoming or outgoing data) technique for reading or writing to a file. If you want to use this approach, the following sections cover some examples, beginning with how to read from a file.

Reading a file

Start a new VB .NET WinForm-style project and then double-click `TextBox` in the WinForms tab of the Toolbox to add it to your `Form1.VB`. Double-click a `Button` to add it also.

Click the `TextBox` in the IDE (Design tab selected) so that you can see its properties in the Properties window. Change its `Multiline` property to `True`.

Now, double-click the `Button` to get to its `Click` event in the code window.

Type this at the top of the code window:

```
Imports System.IO
```

The simplest example looks like this:

```
Private Sub Button1_Click_1(ByVal sender As System.Object,  
    ByVal e As System.EventArgs) Handles  
    Button1.Click  
  
    Dim a As String  
    Dim sr As New StreamReader("e:\test.txt")  
  
    a = sr.ReadLine  
    a += sr.ReadLine  
    sr.Close()  
  
End Sub
```

However, to see a more flexible approach, type this into the `Button`’s `click` event:

```
Private Sub Button1_Click_1(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button1.Click

    Dim strFileName As String = TextBox1.Text

    If (strFileName.Length < 1) Then
        msgbox("Please enter a filename in the TextBox")
        Exit Sub
    End If

    Dim objFilename As FileStream = New FileStream(strFileName,
        FileMode.Open, FileAccess.Read, FileShare.Read)
    Dim objFileRead As StreamReader = New
        StreamReader(objFilename)

    While (objFileRead.Peek() > -1)
        textbox1.Text += objFileRead.ReadLine()
    End While

    objFileRead.Close()
    objFilename.Close()

End Sub
```

Note the `End While` command, which replaces the VB 6 `Wend` command.

Finally, recall that you must add `Imports System.IO` up there at the top of the code window. If System IO is missing, the `FileStream` and `StreamReader` classes will be missing. Or you could create them by adding the name of the assembly (library) with this cumbersome usage:

```
System.IO.StreamReader
```

and

```
System.IO.FileStream
```

Beyond that, you would find the compiler choking when it came to such “undeclared” items as `FileShare`, `FileMode`, and `FileAccess`, which, themselves, would each need the modifier `System.IO`. So, you should play by the rules and simply add `Imports` up top in the code window and avoid all the hassle. Right?

Consider the following line:

```
Dim objFilename As FileStream = New FileStream(strFileName,
    FileMode.Open, FileAccess.Read,
    FileShare.Read)
```

That line would look like the following without the `System.IO` import:

```
Dim objFilename As System.IO.FileStream = New
    System.IO.FileStream(strFileName,
        System.IO.Filemode.Open,
        System.IO.Fileaccess.Read,
        System.IO.Fileshare.Read)
```

Press F5, type a valid filepath to a .TXT file (such as `C:\myfile.txt`) and click the button to see the file's contents in the `TextBox`.

How do you know you're at the end?

When reading from a file, you have to know when you've reached the end. In the previous example, you used the following code:

```
While (objFileRead.Peek() > -1)
```

Alternatively, you can use this:

```
While (objFileRead.PeekChar() <> -1)
```

You can use yet another technique for reading (or writing, using the `FilePut` command). In this case, you test for the end of file with the venerable `EOF` property, End Of File. Also, you use the `FileGet` command to read from a file; in this case, you are reading individual characters. Start a new project and put a `TextBox` on the form. Now type the following:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    Dim objN As New ReadBytes()

    TextBox1.Text = objN.ReadAFile

End Sub

Public Class ReadBytes

    Private strRead As String

    Public Function ReadAFile()
        strRead = ""

        Dim chrHolder As Char
        Dim filenumber As Int16

        filenumber = FreeFile() ' whatever filenumber isn't
            already used

        FileOpen(filenumber, "C:\test.txt", OpenMode.Binary)
```

```
Do While Not EOF(filenumber)
    FileGet(filenumber, chrHolder)
    strRead = strRead & chrHolder
Loop
FileClose(1)

Return strRead

End Function

End Class
```

For your final trick, here's one more way to read from a file. This one does not require a loop:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

Dim objFilename As FileStream = New FileStream("C:\test.txt",
    FileMode.Open, FileAccess.Read, FileShare.Read)
Dim strRdr As New StreamReader(objFilename)

TextBox1.Text = strRdr.ReadToEnd()
TextBox1.SelectionLength = 0 'turn off the selection

End Sub
```

This one relies on the `ReadToEnd` method of the `StreamReader` object. The one kink is that the text that is placed into the `TextBox` is selected (white text on black background). So, to deselect it, you set the `SelectionLength` property to zero.



The `RichTextBox` control has `LoadFile` and `SaveFile` methods, which are well explained in the VB .NET Help feature.

Writing to a file

The code that writes to a file is similar to the previous file-reading example (the one that used the streaming technique). First, be sure to add `Imports System.IO` as described previously.

The simplest approach is as follows:

```
Dim sw As New StreamWriter("test.txt")
sw.WriteLine("My example line.")
sw.WriteLine("A second line.")
sw.Close
```

For a more flexible, advanced example, type the following:

```
Private Sub Button1_Click_1(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button1.Click

    Dim strText As String = TextBox1.Text

    If (strText.Length < 1) Then
        MsgBox("Please type something into the TextBox so we can
            save it.")
    Exit Sub
    Else
        Dim strFileName As String = "C:\MyFile.txt"
        Dim objOpenFile As FileStream = New
            FileStream(strFileName, FileMode.Append,
                FileAccess.Write, FileShare.Read)
        Dim objStreamWriter As StreamWriter = New
            StreamWriter(objOpenFile)

        objStreamWriter.WriteLine(strText)
        objStreamWriter.Close()
        objOpenFile.Close()
    End If
End Sub
```

Because you used the `FileMode.Append` property, each time you run this program new text will be added to any existing text in the file. If you want to overwrite the file, use `FileMode.Create` instead.

Alternatively, you can save to a file by borrowing functionality from the `SaveFileDialog` class (or the `SaveFileDialog` control), like this:

```
Dim sfd As New SaveFileDialog()
Dim dlgResponse As Integer
Dim strFname As String

sfd.DefaultExt = "txt" ' specifies a default extension
sfd.InitialDirectory = "C:"

dlgResponse = sfd.ShowDialog

If dlgResponse = 1 Then
    strFname = sfd.FileName
    msgbox(strFname)
End If
```

(Then add code here to actually save this file.)

Yet another alternative reads and writes individual pieces of data. The size of these pieces is up to you when you define the variable used to write and when you define the read mode (as in `r.ReadByte()` versus `r.ReadBoolean` or `r.ReadInt32`, and so on).

This technique also requires `Imports System.IO`. Here's an example of how to create a file and store bytes into it:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    Dim fs As FileStream = New FileStream("c:\test1.txt",
        FileMode.Create)
    Dim w As BinaryWriter = New BinaryWriter(fs)
    Dim i As Byte

    ' store 14 integers in a file (as bytes)
    For i = 1 To 14
        w.Write(i)
    Next

    w.Close()
    fs.Close()
```

And here's how you read individual bytes from a file:

```
' Create the reader for this particular file:
fs = New FileStream("c:\test1.txt", FileMode.Open,
    FileAccess.Read)
Dim r As New BinaryReader(fs)

' Read the data, in bytes, from the Test1.txt file:
For i = 1 To 14
    Debug.WriteLine(r.ReadByte())
Next i
r.Close()
fs.Close()
```

`BinaryReader` and `BinaryWriter` can read and write information in quite a few different data types. Type this line, and when you type the `.` (period), you see the list of data types:

```
Debug.WriteLine(r.
```

Also note that there are several `FileModes` you can define. The one used in the preceding example breaks with an error if the file already exists.

(To replace an existing file, use `FileMode.Create` instead of `FileMode.CreateNew`.)

Fixed-Length Strings Are Not Permitted

Before VB .NET

When using `Get` or `Put` for random-mode disk file access, you needed to employ strings of a fixed, known length. You could create one by using the following code:

```
A = String(20, " ")
```

This code caused the string variable `A` to have 20 space characters. A more common way to create a fixed-length string, however, was by using `Dim A As String * 20`. When a string was defined to have a fixed length, any attempt to assign a shorter value resulted in the string being padded with spaces and any longer values were truncated to fit.

VB .NET

Fixed-length strings in VB .NET require a different declaration than they required in previous versions of VB. Consider this example:

```
Dim A As String * 20 'This is the pre-VB .NET way
```

That code now changes to the following new declaration syntax:

```
Dim A As VB6.FixedLengthString(20)
```

Additionally, to use this code, you must also use the “compatibility” `Imports` statement:

```
Imports Microsoft.VisualBasic
```

Or you can use the `PadRight` or `PadLeft` methods of the `String` object, like this:

```
Dim n As String = "Title 1"  
n = n.PadRight(75)  
MsgBox(n.Length)
```

Focus

Before VB .NET

If you wanted to force the focus to a particular control, you would use the `SetFocus` method:

```
TextBox1.SetFocus
```

VB .NET

`SetFocus` has been replaced by the `Focus` method:

```
TextBox1.Focus
```

Or you can use this alternative:

```
ActiveControl = TextBox1
```

Focused Property

This new property tells you whether the control has the focus.

Font Property Cannot Directly Be Changed at Runtime

Before VB .NET

Here's an example of how the `Font` property could be used at runtime prior to VB .NET:

```
Sub CommandButton1_Click()  
Label1.Font.Italic = True  
End Sub
```

VB .NET

You can no longer simply change a property of a control's font (its name, size, boldness, and so on). To change these qualities at run time, you have to take an indirect approach by creating a new `Font` object and then assigning it to a control's `Font` property, like this:

```
Private Sub Button1_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button1.Click

    ' Assign a Font object--Name and Size are required.
    Label1.Font = New System.Drawing.Font("Courier New", 20)

    ' Assign additional attributes.

    Label1.Font = New System.Drawing.Font(Label1().Font,
        FontStyle.Bold)
End Sub
```

First, you must assign both name and size (both are required). Then in the next line, you can specify additional properties such as italic or bold, as illustrated in this example.

Inheriting container font properties

In VB .NET, if you change the font properties of the form, all the controls' font properties will also change to match their "parent" container form. However, if you specifically change the font of a control, any additional changes to the form's font properties will not be inherited by the "child" controls. Here's how to adjust all the controls' fonts by simply changing the form:

```
Me.Font = New System.Drawing.Font("Courier New", 20)
Me.Font = New System.Drawing.Font(Me.Font, FontStyle.Italic)
```

Enumerating fonts

In VB 6, you used the `Screen.FontCount` to find out how many fonts were available, and you used the `Screen.Fonts` collection to list (enumerate) them. Now, in VB .NET, there is no `Screen` object. Instead, use the `System.Drawing.FontFamily` object, like this:

```
Dim F As System.Drawing.FontFamily

For Each F In System.Drawing.FontFamily.Families
    Debug.Write(F.Name)
    Debug.WriteLine(" ")
Next
```

Form Events Are Now Called “Base Class Events”

Before VB .NET

If you wanted to have VB type in a form’s event for you, you dropped the list box in the upper-left corner of the code window and selected the Form’s Name (such as Form1). Then you dropped the list box in the upper-right corner of the code window and clicked the event that you wanted to type in. For instance, following these steps and clicking the form’s `KeyPress` event in the list would result in VB typing the following:

```
Sub Form1_KeyPress()  
End Sub
```

VB .NET

To have VB .NET type in a form’s event for you, you drop the list box in the upper-left corner of the code window and select `(Base Class Events)`. However, the newer VB .NET version 2003 changes this: If you are using VB .NET 2003, select `(Form1 Events)`.

Then you drop the list box in the upper-right corner of the code window and click the event that you want to be typed in. Clicking the Base Class Event named `KeyPress` in the top right list results in VB .NET typing the following:

```
Private Sub Form1_KeyPress(ByVal sender As Object, ByVal e As  
    System.Windows.Forms.KeyPressEventArgs) Handles  
    MyBase.KeyPress  
End Sub
```



If you don’t see events or other members automatically listed — such as when you’re creating a new `UserControl` — go to `Tools`⇨`Options`⇨`Text Editor`⇨`Basic`⇨`General` and deselect `Hide Advanced Members`.

Form References (Communication between Forms)

Before VB .NET (from inside the form)

You could reference a form's properties in code inside that form by merely specifying a property (leaving off the name of the form):

```
BackColor = vbBlue
```

Before VB .NET (from outside the form)

If you want to show or adjust properties of controls in one form (by writing programming in a second form), you merely use the outside form's name in your code. For instance, in a `CommandButton_Click` event in `Form1`, you can `Show Form2` and change the `ForeColor` of a `TextBox` on `Form2`, like this:

```
Sub Command1_Click ()  
    Form2.Show  
    Form2.Text1.ForeColor = vbBlue  
End Sub
```

VB .NET (from inside the form)

To reference a form's properties from code inside the form, you must use `Me`:

```
Me.BackColor = Color.Blue
```

VB .NET (from outside the form)

Say that you want to be able to contact `Form2` from within `Form1`. You want to avoid creating clone after clone of `Form2`. If you use the `New` statement willy-nilly all over the place (`Dim FS As New Form2`), you propagate multiple copies of `Form2`, which is not what you want. You don't want lots of windows floating around in the user's Taskbar, all of them clones of the original `Form2`.

Instead, you want to be able to communicate with the single, original Form2 object from Form1. One way to achieve this is to create a public variable in Form1, like this:

```
Public f2 As New Form2

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    f2.Show()
    f2.BackColor = Color.Blue

End Sub
```

Form1 is instantiated first when a VB .NET project executes (by default, it is the “startup object”) in any Windows-style VB .NET project. So, by creating a Public variable that instantiates Form1 (the `New` keyword does that), you can then reference this variable (`f2` here) any time you need to manipulate Form2’s properties or methods from within Form1. It’s now possible for Form1 to be a client of Form2, in other words.

The problem of communicating from Form2 to Form1, however, is somewhat more complex. You cannot use the `New` keyword in Form2 or any other form because that would create a second Form1. Form1 already exists because it is the default startup object.

To create a way of accessing Form1, create an object variable in a module; then assign Form1 to this variable when Form1 is created. The best place to do this assigning is just following the `InitializeComponent()` line in the `Sub New()` constructor for Form1.

Create a module (choose `Project` → `Add Module`). Modules are visible to all the forms in a project. In this module, you define a public variable that points to Form1. In the module, type this:

```
Module Module1

    Public f1 As Form1()

End Module
```

Notice that the `New` keyword was not employed here. You are merely creating an object variable that will be assigned to point to Form1. Click the + next to “Windows Form Designer generated code” in Form1’s code window. Locate Form1’s constructor (`Public Sub New`) and, just below the `InitializeComponent()` line, type this:

```
InitializeComponent()

F1 = Me
```

This assigns `Me` (Form1, in this case) to the public variable `F1`. Now, whenever you need to communicate with Form1 from any other form, you can use `F1`. For example, in Form2's Load event, you can have this code:

```
Private Sub Form2_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    F1.BackColor = Color.Brown

End Sub
```

Formatting Strings

In VB .NET, you can format strings by using a set of character symbols that represents typical formatting styles.

The following code gives a result of \$44,556,677.00:

```
Dim i As Integer = 44556677
Dim s As String = i.ToString("c") ' use the "c" for currency
    format
Debug.Write(s)
```

There is also `d` for decimal, `e` for exponential, `f` for fixed point, `n` for number, `p` for percent, `x` for hex, and a whole variety of other symbols you can look up in VB .NET Help. To look them up, choose Help⇨Search and then type the following line into the address field:

```
ms-help://MS.VSCC/MS.MSDNVS/cpguide/html/
    cpconformattingoverview.htm
```

Here's an example showing how various formatting symbols can be combined:

```
Dim d As Double = 44.556677
Dim s As String = d.ToString("%#.##")
Debug.Write(s)
```

This results in:

```
%4455.67
```

If you're using a `StringBuilder` object, you can employ its `AppendFormat` method by using one of the formatting symbols.

Frame Becomes Group Box

Before VB .NET

The Frame was the control used to create subdivisions on forms, to optionally label those subdivisions, and to enclose a set of option buttons (so that when the user clicked one button, the previously selected button was unselected).

In VB .NET

The Frame is now named the Group Box, but it behaves the same way as the Frame did.

Function Return Alternative

Before VB .NET

In traditional VB, a function returned a value to the caller by simply assigning that value to the function's name. For example, this function multiplied a number by 2 and then returned the result to the caller:

```
Public Function DoubleIt(passednumber As Integer)

    passednumber = passednumber * 2

    DoubleIt = passednumber
    'assign the result to the function's name
    'so it gets passed back to the caller

End Function
```

VB .NET

It's not mandatory, but if you want to use C-like syntax, you can now use the following way of returning a value to the caller:

```
Public Function DoubleIt(ByVal passednumber As Integer) As Integer
    passednumber = passednumber * 2
    'use the Return statement to pass the result
    'back to the caller
    Return (passednumber)
End Function
```

I don't know about you, but I view this syntax as a real improvement over the way that VB has always returned function values. It's clearer and easier to read. I confess that some C-language diction and syntax are, to me, not as readable as traditional VB. But this one is an exception.



In the following function, the `If . . . Then` test will never execute:

```
Public Function DoubleIt(ByVal passednumber As Integer) As Integer
    passednumber = passednumber * 2
    Return (passednumber)

    If passednumber > 50 Then MsgBox ("Greater than 50")
End Function
```

The `If . . . Then` line comes after the `Return`, and anything following `Return` in a function is not executed. `Return` causes an immediate `Exit Function`.

Garbage Collection Debate

Before VB .NET

One of the most controversial changes in VB .NET is the way that objects are terminated. As defined by the Microsoft COM framework, when an object is instantiated (brought into being), it is automatically kept track of, and when it is no longer needed by the program, its `Terminate` event is triggered. (You can kill an object in this way: `Set MyObject = Nothing`.)

An object's `Terminate` event allows you to do any needed shutdown house-keeping chores (by putting such code into the `Terminate` event), and then the object is destroyed. This keeps things clean and, above all, frees up memory. (The use of a `Terminate` event in this fashion is called *deterministic finalization*.)

VB .NET

VB .NET does not keep track of objects in the COM fashion. Instead, .NET uses a *garbage collection* (GC) technique that checks things from time to time during a VB .NET program's execution. It checks its list of object references, and if an object is no longer in use, GC deletes it and thereby frees memory. This means that you, the programmer, cannot know when an object is destroyed, nor the order in which objects in an object hierarchy are destroyed. You cannot rely on a `Terminate` event.

If your VB 6 and older source code relies on an object's `Terminate` event or a predictable order of termination, you'll have to rewrite that object to give it a procedure that shuts it down. And if you need to keep a count of references, you also need to give it a procedure that opens the object and deals with this reference counting.

Technically, the garbage collection approach can prevent some problems, such as memory leaks. And it also permits faster performance. However, many programmers (mostly C programmers, actually) are upset because the garbage collection approach not only requires quite a bit of rewriting to port older programs to VB .NET, but it also means that the programmer, not the computer, has the burden of adding open and close procedures for his or her objects. This is potentially a serious source of errors because the object no longer automatically handles its own termination. (Clients, users of the object, must specifically invoke the `Close` event and ensure that the event doesn't itself spawn any unintended side effects — in other words, clients must error trap.) However, for VB programmers, it's best to just relax and let VB .NET handle this issue.

Global Is Gone

Before VB .NET

VB has had a `Global` command for years. It declares a variable with the widest possible scope — application-wide. Any form or module in the application can see and manipulate a `Global` variable. `Global` is no longer in the language.

VB .NET

The `Public` command is the equivalent of `Global`. Use `Public` instead.

GoSub, On . . . GoSub, and On . . . GoTo Are No More

Before VB .NET

The various `GoSub` and `GoTo` structures have been used in traditional VB for error trapping or for quick, convenient execution of a task that requires many parameters. `GoSub` or `GoTo` commands must be used within a given procedure. Therefore, if you have some code that you `GoSub` to, it would not be necessary to “pass” parameters to this task (the variables would all have local scope, and therefore would be usable by your task).

The `On . . .` commands were also sometimes used instead of `Select . . . Case` or `If` branching situations.

VB .NET

These three commands are removed from the language.

Handle Property

This new property gives you the window handle for this control.

Handles

In VB 6 and earlier, each event of each object (or control) was unique. When the user clicked a particular button, that button’s unique event procedure was triggered. In VB .NET, each event procedure declaration ends with a `Handles` command that specifies which event or events that procedure responds to. In other words, you can create a single procedure that responds to multiple different object events.

To put it another way, in VB .NET, an event can “handle” (have code that responds to) whatever event (or multiple events) that you want it to handle. The actual sub name (such as `Button1_Click`) that you give to an event procedure is functionally irrelevant. You could call it `Bxteen44z_Click` if you want. It would still be the click event for `Button1` no matter what you named this procedure, as long as you provide the name `Button1` following the `Handles` command.

In other words, the following is a legitimate event where you write code to deal with `Button1`'s `Click`:

```
Private Sub Francoise_Click(ByVal sender As System.Object,  
    ByVal e As System.EventArgs) Handles Button1.Click
```

The `Handles Button1.Click` code does the trick.

You can even gang up several events into one, like this:

```
Private Sub cmd_Click(ByVal sender As System.Object, ByVal e  
    As System.EventArgs) Handles cmdOK.Click,  
    cmdApply.Click, cmdCancel.Click
```

For a working example of this ganging up, see the section in this appendix titled “Control Arrays Are Gone.”

Runtime handles

If you want to get really fancy, you can attach or detach an object's events to a procedure during run time. The following example illustrates this.

Put three buttons on a form. Now create this procedure to handle all three of these buttons' `Click` events:

```
Private Sub AllTheButtonClicks(ByVal sender As Object, ByVal  
    e As System.EventArgs) Handles Button1.Click,  
    Button2.Click, Button3.Click  
  
    RemoveHandler Button1.Click, AddressOf AllTheButtonClicks  
  
    MsgBox(sender.ToString)  
  
End Sub
```

Press `F5` to run this little program and click `Button1`. The `MsgBox` appears. Click `Button1` again and nothing happens because `Button1`'s `Click` event has been detached — so this procedure no longer responds to any clicking of

Button1, Button2 and Button3 still trigger this procedure. To restore (or create for the first time) a procedure's ability to handle an object's events, use the `AddHandler` command, like this:

```
AddHandler Button1.Click, AddressOf AllTheButtonClicks
```

HasChildren Property

This new property tells you whether the control contains any child controls.

Height

See “Left, Top, Height, Width, and Move All Change.”

Help System Fails

From time to time, you may notice that when you look up an entry (using Index, Search, it doesn't matter) in the VB .NET Help utility, you get an error message in the right view pane:

```
The page cannot be displayed
The page you are looking for is currently unavailable. The
    Web site might be experiencing technical
    difficulties, or you may need to adjust your
    browser settings.
Etc.
```

It's the usual browser error message when you travel to an Internet address that no longer exists.

Or, you may simply see a blank right pane, no error message, and no help information.

The most likely cause for this problem is a filled browser cache. To cure the problem, open Internet Explorer, and choose Tools⇨Internet Options. On the General tab (the default) in this dialog box, click the Delete Files button under Temporary Internet Files. These files will be deleted (which can take several minutes). Then retry Help, and it should work fine once again.

Help (Using)

The .NET Help system can be used in a variety of ways to assist you when creating VB .NET projects. It's a significant resource, and it's becoming more useful as time goes on (more helpful descriptions, more bug-free source code examples).

IDE: The VB .NET Editor

The VB .NET IDE (Integrated Development Environment) is packed with features and options. You usually work with a solution, which can contain one or more individual projects. You can switch between projects, or right-click project names to adjust their properties within the Solution Explorer (Ctrl+R).

You get used to the IDE and find your own preferred layout. My main piece of advice is to right-click the title bar of all the various windows and choose Auto Hide. That way, the Properties window, Toolbox, Solution Explorer, output window, and so on are all quickly available (because they're tabs along the edges of your IDE). But they don't cover up the primary work surface: the Design and code window. You may need to set the Tabbed Documents option in the Tools⇨Options⇨Environment⇨General menu.

And while you're looking at the Environment section of the Options menu, you may want to click the Help section and switch from the default to External Help. That way, you get a full, separate window for your help searches rather than letting VB .NET try to fit help windows into your IDE.

Under Tools⇨Options⇨Text Editor⇨All Languages⇨General, you want to select Auto List Members and Parameter Information, unless you have a photographic memory and have read through the entire .NET class documentation. These two options are, for most of us, essential: They display the properties and methods (with their arguments and parameters) of each object as you type in your source code. Also, while you're on this page in the dialog box, uncheck the Hide Advanced Members. That way you get to see everything.

Implements Changes

Before VB .NET

VB 6 and earlier versions could inherit classes using the `Implements` command, but they could not create interfaces directly.

VB .NET

In VB .NET, you can give an inherited object's members (its methods and properties) any names you wish. The `Implements` command can be followed by a comma-separated list of names. (Typically, you would use only a single name, but you can use multiple names.)

The names you use in this fashion must be fully qualified, which means you need to supply the interface name, a period, and then the name of the member (method or property) that you want to be implemented. (VB prior to VB .NET used an underscore for the punctuation:

`InterfaceName_MethodName`.)

Imports

The new VB .NET `Imports` command doesn't import anything. Based on its name, you're likely to assume that it adds a library of functions to your project. You'll probably think it's adding a "Reference" to a dynamic link library (DLL). (By the way, the term *DLL* isn't used anymore; it's now called an *assembly*. A given assembly can contain multiple namespaces. In `System.Drawing.Color`, `System.Drawing` is an assembly and `Color` is a namespace within that assembly.)

Anyway, when you use `Imports`, all that happens is that it permits you to write shorter, more convenient source code. After importing, you can then leave out the namespace when referring to a function or object that sits inside the namespace referenced by that `Imports` statement. For example, include the following line at the top of your code window:

```
Imports System.Drawing.Color
```

With that line, you can then use the following shortcut when naming one of the VB .NET color constants:

```
Me.BackColor = Black
```

Or, if you don't use any `Imports` statement, you must specify the `Color` namespace within your source code whenever you use the `Black` constant:

```
Me.BackColor = Color.Black
```

Similarly, the VB .NET control characters (such as `CrLf`, which is used in a `MsgBox` to force the message text to appear down one line) require the `ControlChars` namespace:

```
Dim s As String = "This line."  
s += ControlChars.CrLf + "Next line."  
Debug.Write(s)
```

A namespace is the new .NET term for a group of functions that are stored together (as a logical unit) within an assembly. Recall that there can be more than one namespace in a given assembly (a given code library).

You put `Imports` statements at the very top of your code window. When you do, you can thereafter simply refer in your source code to objects and functions (methods) residing in whatever your `Imports` statement specifies. VB .NET will be able to tell in which code library namespace the function is located — because you specified the namespace’s name in the `Imports` statement.

Here’s an example. Let’s say that you want to write the following code that deletes a directory on the hard drive:

```
Public Function DestroyDirectory()  
  
    Dim objDir As New DirectoryInfo("C:\TestDir")  
    Try  
        objDir.Delete(True)  
    Catch  
        Throw New Exception("Failed to delete")  
    End Try  
  
End Function
```

If you simply type this into the VB .NET code window, it rejects the `DirectoryInfo` object because it doesn’t recognize it. You can fix this problem by specifying the correct namespace adjectives in your source code line, as follows:

```
Dim objDir As New System.IO.DirectoryInfo("C:\TestDir")
```

That `System.IO` specifies the namespace adjectives so that VB .NET knows in which library to find the `DirectoryInfo` object and its methods. However, if you are going to be using `DirectoryInfo` or other IO functions several times, you may not want to keep having to type in that `System.IO` reference over and over. You can shorten your source code by putting this `Imports` statement at the top:

```
Imports System.IO
```

Now you can shorten the example line, merely using the `DirectoryInfo` command, without “qualifying” it by specifying the namespace where this command can be found. Here’s how you can shorten this line of code:

```
Dim objDir As New DirectoryInfo("C:\TestDir")
```

With no `Imports` statement, you must “fully qualify” (use adjectives).

In earlier versions of VB, all the commands in the language were always available. You never needed to specify a namespace (there was no such concept as namespace).

In VB .NET, some namespaces are loaded in by default, so you need not use `Imports` for them. However, if you are trying to use a valid VB .NET command (such as `DirectoryInfo`) and you get the error message `Type is not defined or something like it`, consider that you might have to `Imports` a namespace for that command. (For example, when you work with certain database commands, you have to use `Imports`.)

To see the available namespaces, go to the top of your code window and type the following:

```
Imports System.
```

As soon as you type the `.` (period), you see a list. If you’re working with databases, try the following:

```
Imports System.Data.
```

Now, when you type the second period (after `Data`), you see another list of namespaces. If you are going to use SQL Server databases, you may then want to `Imports` all namespaces beginning with `SQL` (each namespace `Imports` statement must be on a separate line in your code window):

```
Imports System.Data.SqlClient
Imports System.Data.SqlDbType
Imports System.Data.SqlTypes
```

In this situation, however, your best bet is to get a good book and just follow its advice about which namespaces you need to `Imports`. See this book for information on namespaces to use for database, graphics, security, and other project categories. Otherwise, you just have to learn by trial and error for yourself.



If you look up a class in Help, such as `DirectoryInfo`, you frequently find a reference to its namespace down at the bottom of the right pane (be sure to scroll down as necessary to see if the namespace is mentioned in a section titled “Requirements”).

Why namespaces?

Why are they using namespaces? It's a clerical thing. It prevents "name collisions" if there are two identically named functions that do two different things (and each function is in a different code library). Which one does the programmer mean to use? By adding `Imports` statements at the top of the code window (or by qualifying the function name each time it appears within the source code), you are saying: "Use the function in this namespace — not the other one that is in some other namespace." (You can even have multiple namespaces within a single assembly — a single library, a single DLL.)

Sometimes, you may find that you need to actually add a library of code to a VB .NET project. In other words, a function or functions that you want to use aren't part of the default set of libraries added to a VB .NET project. (To see which libraries are added by default, right-click the name of your project in Solution Explorer, choose Properties, and then select Imports in the Properties dialog box. To add a namespace to your VB .NET project, choose Project → Add Reference.)

What happens, you may say, if you use `Imports` to name two namespaces that have a pair of identically named objects? Now you will have a name collision — no doubt about it. `Imports` doesn't help you distinguish between these objects because you've used `Imports` for both namespaces.

For example, assume that you write code for a custom cursor class and put it into a namespace called `NewCursors` within your library `MyObjects`. Then you `Imports` it:

```
Imports MyObjects.NewCursors
```

And you also `Imports` the standard Windows objects namespace (which includes controls such as the `TextBox`, as well as other items, such as cursors):

```
Imports System.Windows.Forms
```



You need not actually use `Imports System.Windows.Forms`; it's already imported by default by VB .NET, but I'm just showing you an example here.

Now, you have two namespaces in your project, each containing a (different) cursor class. The only way you can distinguish them is to forget about the `Imports` trick and just use the long form (fully qualified) whenever you use the `Cursor` object:

```
Dim n As New System.Windows.Forms.Cursor("nac.ico")
```

or

```
Dim n As New MyObjects.NewCursors.Cursor("nac.ico")
```

Practical advice about namespaces

Don't worry about using Imports to add whatever namespaces you may need: Namespaces neither increase the size of your executable (.EXE) program, nor do they slow execution.

VB .NET is a huge language. It includes more than 60 .NET assemblies, containing the hundreds of .NET namespaces. Each namespace contains many classes. In turn, these classes have multiple members (methods you can employ, properties you can read or change). Also, many of the methods are overloaded: They often permit you to provide a variety of arguments to make them behave in different ways. As you can see, there are hundreds of thousands of commands and variations of those commands in this language.

What to do? You can learn the important tools quickly — file I/O, printing, useful constants, interacting with the user, debugging, and so on. Most of these major programming techniques can be found in this appendix, as a matter of fact. Also, you can use VB .NET Help's Search and Index features as necessary. To see the format, syntax, punctuation, and, sometimes, code examples of the many classes, try this approach. Run the Help Index and then type the following into the Look For field (cutting and pasting doesn't work; you must type this in):

```
system.drawing.
```

You then see the massive number of classes listed in the left pane. Click on any of these classes to see its members and other information about how to use it. Each page contains hyperlinks that take you to more specific information about particular members.

The more you work with VB .NET, the more you learn about which namespaces you need to Imports. Often, you don't need to Imports any because the most common namespaces are automatically imported by VB .NET when you create a new project. These seven are imported by default: Microsoft.VisualBasic (a "compatibility" namespace, permitting you to use most VB 6 constants and functions, such as InStr rather than the new .NET equivalent, IndexOf); System; System.Collections; System.Data; System.Diagnostics; System.Drawing; and System.Windows.Forms.

You may remember some traditional VB constants, such as vbBlue. Well, they are not included in the Microsoft.VisualBasic compatibility namespace that's automatically imported by default. Use the new VB .NET constants instead:

```
Me.BackColor = Color.Black
```

The VB .NET color constants are in the `System.Drawing.Color` namespace, but you only have to use `Color` because `System.Drawing` is one of the seven automatically imported default assemblies. And, if you wish, you can avoid having to specify `Color` in your code by adding the following line at the top of your code window:

```
Imports System.Drawing.Color
```

The Object Browser

Press F2 or Ctrl+Alt+J, and the Object Browser opens. Browse through the various assemblies and their namespaces to see what's available — which constants, properties, and methods are contained within which namespaces.

Inheritance

Some VB programmers have long wanted the language to have true inheritance, one of the big three features of object-oriented programming (the other two being encapsulation and polymorphism). Well, they got their wish.

Here's an example showing you how to inherit from the built-in `TextBox` class. In this example, you have a problem. Your daughter cannot speak or write without using the term “really” every few words. You want your `TextBox` to have a method that deletes all instances of the word “really” at the click of a button. Here's how:

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Dim t As New NewTextBox()
    Private Sub Form1_Load(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles MyBase.Load

        t.Multiline = True
        t.Size = New System.Drawing.Size(130, 130)
        t.Location = New System.Drawing.Point(10, 50)
        t.Name = "SpecialTextBox"
        Me.Controls.Add(t)
    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles Button1.Click
        t.RemoveReally()
    End Sub
End Class
```

```
Public Class NewTextBox
    Inherits System.Windows.Forms.TextBox

    Public Sub RemoveReally()

        Dim s As System.Text.StringBuilder
        s = New System.Text.StringBuilder(Me.Text)

        s.Replace("really", "") ' get rid of any really's.
        Me.Text = s.ToString

    End Sub

End Class
```

The new `TextBox` class that you created inherits all the members (properties, events, and methods) in the ordinary VB .NET `TextBox`. But you define a new method, called `RemoveReally`, which replaces all instances of “really” in its `Text` property.

In the `Page_Load` event, you specify various properties of the new `TextBox`. Run this program by pressing F5 and then type in some text with the word “really” in various places. Then click the Button that invokes the `RemoveReally` method.

Initializers

VB .NET permits you to simultaneously declare a variable or array and assign a value to it:

```
Dim n As Integer = 12
```

or:

```
Dim strVisitors() As String = {"Morris", "Susan"}
```

Also, if you are using the latest version, VB .NET 2003, you can even declare a loop counter variable within the line that starts the loop, like this:

```
For i As Integer = 0 To 10
```

instead of the traditional style:

```
Dim i as Integer
For i = 0 To 10
```

Also see “Dim and ReDim: New Rules.”

Instantiation

See “Object Instantiation.”

InStr

See “String Functions Are Now Methods.”

Integers and Long Double in Size

Before VB .NET

The `Integer` data type was 16 bits large, and the `Long` data type was 32 bits large.

VB .NET

Now `Integer` and `Long` are twice as big: `Integer` is 32 bits large and `Long` is 64 bits (it’s an `Integer`, too, with no fractional control and no decimal point). If your program needs to use a 16-bit integer, use the new `Short` type.

So if you’re translating pre-.NET VB code, you need to change any `As Integer` or `CInt` commands to `As Short` and `CShort`, respectively. Similarly, `As Long` and `CLng` now must be changed to `As Integer` and `CInt`. See also “Data Types.”

IsEmpty

In VB 6, if a variant variable had previously been declared in the source code, but not yet given any value (not initialized), you could check to see if the variant contained empty as its “value” (`IsEmpty`). The `IsEmpty` command is not available in VB .NET, nor is the `Variant` type itself.

In VB .NET, the default variable type is the `Object`. An `Object` variable has a default value of nothing.

IsNull

In VB 6, you could query to see if a variable held a `Null` by using the `IsNull` command. This command has been changed to `IsDBNull`, and the `Null` value is now `DBNull` and can only be used with database fields, not in other situations (such as string concatenation).

KeyPress

The following example shows how to use the `e.KeyChar` object in VB .NET to figure out which key the user pressed. You compare it to the `Keys` enumeration (a list of built-in constants that can be used to identify keypresses). The following code checks to see if the user presses the Enter key:

```
Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e
    As System.Windows.Forms.KeyPressEventArgs)
    Handles TextBox1.KeyPress

    If Asc(e.KeyChar) = Keys.Enter Then
        MsgBox("ENTER")
    End If

End Sub
```

The following example shows how to detect if the user presses Ctrl+N on the form (to set the Form's `KeyPreview` property to `True`):

```
Private Sub Form1_KeyDown(ByVal sender As Object, ByVal e As
    System.Windows.Forms.KeyEventArgs) Handles
    MyBase.KeyDown

    If e.KeyCode = Keys.N And e.Control = True Then
        'they pressed CTRL+N

        searchnext() 'respond to this key combination
        Exit Sub
    End If

End Sub
```

LCase and UCase Change

Many traditional VB string functions — `LCase`, `UCase`, `LTrim`, `RTrim`, and `Trim` — have been changed to VB .NET methods. So, the following classic function syntax has changed:

```
Dim s As String = "Bob"  
MsgBox(LCase(s))
```

You now use the syntax appropriate to a method, as follows:

```
Dim s As String = "Bob"  
MsgBox(s.ToLower)
```

LCase, UCase, LTrim, RTrim, and Trim have been changed to, respectively, ToLower, ToUpper, TrimStart, TrimEnd, and Trim.

See “String Functions Are Now Methods” for additional information.

Left (LeftStr or Left\$) Must Be “Qualified”

See “Right Property.” Also see “String Functions Are Now Methods.”

Left, Top, Height, Width, and Move All Change

Before VB .NET

You could specify a new position or size for a control like this:

```
CommandButton1.Left = 122  
CommandButton1.Height = 163
```

VB .NET

In VB .NET, you have several ways to position or size items using code.

In the following code, the `With...End With` structure is used to provide references to other controls, in order to specify size and position:

```
With Button1
    .Left = btnSearch.Left
    .Top = txtTitle.Top
    .Width = btnSearch.Width
    .Height = txtTitle.Height + txtDesc.Height + 17
End With
```

Or you can use the traditional VB property names:

```
btnSearch.Left = 222
btnsearch.top = 155
```

However, you can no longer use `Left` or `Top` to position a form. Instead, use this:

```
Me.Location = New Point(10, 12)
```

You can use the new (preferred) CLR-compatible (Common Language Runtime) property names in the Properties window:

```
Location.X, Location.Y, Size.Width and Size.Height
```

Or you can “qualify” your code, which means that you add various library references and object references to adjust position or size. (See “Imports” for more on namespaces.) Here are some examples for you to follow if you want to change these qualities while your application is running:

```
Button1.Location = New System.Drawing.Point(256, 160)
Button1.Size = New System.Drawing.Size(120, 32)
```

Alternatively, you can use code like this if the `System.Drawing` namespace is already available to your code (because at the top of your code window, you include the line `Imports System.Drawing`):

```
Button1.Location = New Point(100, 125)
```

Line Control (And Command) Are Gone

See “Circle, Line, Shape, PSet, and Point All Changed.”

LinkLabel Control

This new control displays an Internet URL, and then you can use its `Clicked` property to run Internet Explorer and automatically go to the Web site described in the `LinkLabel`’s `Text` property:

```
Private Sub LinkLabel1_LinkClicked(ByVal sender As
    System.Object, ByVal e As System.Windows.Forms
        .LinkLabelLinkClickedEventArgs) Handles
        LinkLabel1.LinkClicked

    System.Diagnostics.Process.Start("http://www.cnn.com")

End Sub
```

ListBox (And Others) Become Objectified

Before VB .NET

You could write code like this:

```
Listbox1.AddItem(n)
```

VB .NET

If you try using the `AddItem` method of a `ListBox` in VB .NET, you get an error message informing you that `AddItem` “is not a member of the `System.Windows.Forms.ListBox`” object. Well, excuuuuuse me! `AddItem` used to be a member!

Little kinks like this pop up here and there as you’re adjusting yourself to VB .NET programming. You then should look up the control in VB .NET Help to see what the proper diction is to accomplish your goal. In this example, you must use the `Add` method of the `ListBox` control’s `Items` collection. Here’s the solution:

```
Listbox1.Items.Add(n)
```

Also, if you want to clear the contents of a `ListBox` (or similar control), you must now use the `Clear` method of the `Items` collection, like this:

```
Listbox1.Items.Clear()
```

The `ListBox` control in previous versions of VB had its own `Clear` method (`List1.Clear`).

Another change in terminology involves the way that you access which item the user clicked within a `ListBox`. In traditional VB, you get the clicked item's index number like this:

```
Dim SelectedItem as Integer
SelectedItem = ListBox1.List(ListBox1.ListIndex)
```

But in VB .NET, you use the following code:

```
Dim SelectedItem as Integer
SelectedItem = ListBox1.Items(ListBox1.SelectedIndex)
```

Or, to get the actual string that was clicked (not the index), use the following code:

```
Dim SelectedItem as String
SelectedItem = ListBox1.Items(ListBox1.SelectedItem)
```



You probably assume that if you use an `Add` method to add items, there is a corresponding `Remove` method to remove items. Well, Dude, there used to be. The powers that control the language decided in their mysterious way to change this method's name to `RemoveAt`:

```
ListBox1.Items.RemoveAt (ListBox1.SelectedIndex)
```

Other changes to the `ListBox` include: It can now have multiple columns; its items are objects, not strings; and you can display any property of the objects in the `ListBox`. The following example shows how to list the `Name` properties of all the controls currently on the form in a two-column `ListBox`:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    Dim n As Integer = Me.Controls.Count
    Dim i As Integer

    With ListBox1

        .MultiColumn = True
        'make width of each column less than half of listbox width
        .ColumnWidth = (.Width \ 2) - 10

    End With

    For i = 0 To n - 1
        ListBox1.Items.Add(Me.Controls(i).Name)
    Next

End Sub
```

LoadPicture Replaced

Before VB .NET

To put a graphic into a VB PictureBox in VB 6, you used this code:

```
Set Picture1.Picture = LoadPicture("C:\Graphics\MyDog.jpg")
```

VB .NET

LoadPicture has been replaced with the following code:

```
PictureBox1.Image = Image.FromFile("C:\Graphics\MyDog.jpg")
```

Notice the various changes:

- ✓ The default name for a PictureBox is no longer `Picture1` but rather `PictureBox1`.
- ✓ The `Picture` property has been changed to an `Image` property.
- ✓ Rather than `LoadPicture`, you now use `Image.FromFile`.

Location Property

This new property specifies the coordinates of the upper-left corner of the control, relative to the upper-left corner of its container (usually the form).

LSet and RSet Are Gone

If you need to pad strings, use the new `PadRight` and `PadLeft` commands.

You cannot use `LSet` or `RSet` to assign one data type to another.

For additional information, see “User-Defined Type Changes to Structure” in this appendix.

LTrim

See “LCase and UCase Change” and “String Functions Are Now Methods.”

Math Functions Change (ATN, SGN, ABS, and So On)

The various VB 6 math functions (such as `Atn`, `Sgn`, and `Sqr`) have been replaced by (often) differently named functions in VB .NET: `Atan`, `Sign`, and `Sqrt`, for example. What’s more, they are now methods of the `System.Math` class. So, you must use them like this:

```
Private Sub Button1_Click_1(ByVal sender As System.Object,  
    ByVal e As System.EventArgs) Handles  
    Button1.Click  
  
    Dim x As Double  
  
    x = System.Math.Sqrt(16)  
    MsgBox(x)  
  
End Sub
```

The obsolete VB 6 version looked like this:

```
MsgBox (Sqr(16))
```

Mathematical functionality is supplied by classes in the `System` namespace. The `Atn`, `Sgn`, and `Sqr` functions are replaced by the `Atan`, `Sign`, and `Sqrt` methods of the `Math` class. `System.Math` offers comprehensive mathematical support with its methods and fields.

Mid

See “String Functions Are Now Methods.”

Move

See “Left, Top, Height, Width, and Move All Change.”

Name Property Becomes Dynamic

In VB .NET, the `Name` property can be changed during run time. This is useful when you create the VB .NET version of control arrays. See “Control Arrays Are Gone.”

Namespace

See “Imports.”

Null

See “IsNull.”

Object Instantiation

Before VB .NET

A common format for instantiating (creating) an object in VB 6 and earlier is as follows:

```
Dim DataCon as New ADODB.Connection
```

However, some people argued that this wasn't good programming because it added overhead (inefficiency) to the execution (never mind why). Instead, you were asked to use this version:

```
Dim DataCon as ADODB.Connection  
Set DataCon = New ADODB.Connection
```

VB .NET

The `Set` command has been removed from the language in VB .NET.

All three of the following syntaxes do the same thing in VB .NET (instantiate an object — bring it into existence):

```
Dim DataCon as New ADODB.Connection

Dim DataCon as ADODB.Connection = New ADODB.Connection

Dim DataCon as ADODB.Connection
DataCon = New ADODB.Connection
```

However, the third version gives you a bit more flexibility because the object is not created in the declare (`Dim`) statement. It's created in the line where you use the `New` command. This way, you can declare the object variable with `Dim`, but delay the actual instantiation of the object (with `New`) until you actually need that object. If this distinction means something to your programming, go ahead and use the most verbose format (version three in this example).

ObjPtr, VarPtr, and StrPtr Are Removed

Before VB .NET

Although you would search for them in vain in the VB Help engine, there were three “secret” commands in VB: `ObjPtr`, `VarPtr`, and `StrPtr`. You could use them to get a variable's address in memory. This isn't officially permitted in VB, so it's undocumented. However, using such commands, some extreme programmer jockeys were able to speed up their applications or add functionality that VB itself didn't directly offer. (Interestingly, although these three functions are missing from Help, their syntax is displayed to you if you've got the Auto Quick Info option set in the Tools⇨Options menu.)

Here's an example:

```
Private Sub Form_Load()

    Dim books As String
    books = "Dickens"
    n = StrPtr(books)
    MsgBox n 'display the variable's address in memory

End Sub
```

VB .NET

These functions have been removed. Like several other elements in VB 6 and earlier versions, these commands have simply been eliminated from the language (with no substitutions to provide a replacement for their functionality).

Optional Parameter-Passing Tightened

Before VB .NET

In VB 6 and earlier, you could freely change a default value for parameters in a procedure. For instance, if no parameter is passed to this sub, it will default to the value 101 for the `Books` parameter:

```
Private Sub CountBooks(Optional Books As Integer = 101)
```

Because VB used to check parameters and employ any necessary default parameters at run time, it was possible to change a control to use a different default value and VB would respect that new value.

What's more, you could simply declare a parameter as `Optional` and not provide any default whatsoever, if you wished:

```
Private Sub CountBooks(Optional Books As Integer)
```

And, if the optional parameter is a `Variant` type, you could have used the `IsMissing` command to determine whether the parameter was passed at all.

VB .NET

VB .NET compiles any needed default values — and the result becomes hard-wired into the compilation. If the control is changed and a procedure is given new default parameter values, the original defaults will still be used. Any new defaults cannot take effect unless you completely recompile the program. The virtue of this approach is that it speeds things up at run time. Also, VB .NET offers an alternative to optional parameters, which is known as *overloading*. See the section titled “Overloaded Functions, Properties, and Methods (Overloading)” in this appendix.

In VB .NET, every optional parameter must declare a default value (it will be passed to the procedure if the calling code does not supply that parameter). The `IsMissing` command is no longer part of the VB language. Note, too, that you must include the `ByVal` command:

```
Private Sub CountBooks(Optional ByVal Books As Integer = 101)
```

OptionButton Is Renamed RadioButton

The heading says it all!

Option Strict and Option Explicit

Before VB .NET

In VB 6, you were not required to explicitly declare variables. You could simply assign a value to a variable, and that caused the variable to come into existence:

```
MyVariable = 12
```

Also, you were permitted to forget about variable types and let VB decide for you which type was required at any given place in the code, based on the context. For example, if `MyVariable` contained numeric 12, yet you wanted to display it in a `TextBox`, VB 6 would automatically transform this numeric integer data type into a text string type:

```
Text1 = MyVariable
```

This line would cause `Text1` to display the digit characters 12.

VB .NET

Many programmers believe that all variables should be explicitly declared. It's not enough to simply assign some value to a variable; you should declare it and declare its type as well:

```
Dim MyVariable As Integer  
MyVariable = 12
```

Not only that, if you want to transform (cast, or coerce) a variable of one type into another type — such as changing an `Integer` type into a `String` type — you should specifically do that transforming in your source code:

```
TextBox1.Text = CStr(MyVariable)
```

The `CStr` command is one of several commands that begin with `C` (`CInt`, `CByte`, and so on) that cast one variable type into another.

In VB .NET, if you attempt to use a variable before explicitly declaring it, VB .NET will inform you that you've made an error, that the variable name has not been declared. If you want to turn off this error message and use undeclared variables, type this line at the very top, on the first line, in your Code window:

```
Option Explicit Off
```

However, you can avoid some kinds of bugs by explicitly declaring all your variables, and it doesn't take too much extra time to declare them.

Although VB .NET enforces `Option Explicit` by default, it does not enforce `Option Strict` by default. `Option Strict` governs whether an error message will be generated if you don't cast variables. You can use this line of code, even though the `TextBox` displays a string and `MyVariable` was declared as an `Integer` variable type:

```
TextBox1.Text = MyVariable
```

This is perfectly legal in VB .NET. No error message will be displayed when you execute this code. However, some programmers insist that casting is important. If you are one of those, you can enforce casting by typing this line at the top of your code window:

```
Option Strict On
```

Now, try to execute the following line:

```
TextBox1.Text = MyVariable
```

The VB .NET compiler does not like that line and displays the following error message:

```
Option strict disallows implicit conversions from  
System.Integer to System.String.
```

`Option Strict` does permit VB .NET to handle a few "safe" data type conversions itself. For instance, changing a 16-bit integer to a 32-bit integer cannot result in any loss of data, so it is permitted to take place automatically within VB .NET — no need for the programmer to explicitly do this conversion with special source code. The reverse, though (changing a 32-bit integer into a 16-bit version), is not permitted to take place automatically. You, the programmer, must use one of the data-conversion commands (see "Converting Data Types" in this appendix). Likewise, a floating-point data type (which may hold a value such as 12.335) is not permitted to automatically convert to an integer (which would strip off the .335 fractional portion of the number because integers have no decimal point). Is converting from an integer to a floating-point type permitted? Yes, because that is "safe."

Optional Operation Shorthand

Some programmers (users of the C language and its offspring) prefer a slightly abbreviated form of arithmetic, incrementation, and text concatenation. To some VB programmers, it can seem harder to read at first. This new syntax is optional, so you can stick with the traditional VB syntax if you prefer. But it can save some time and effort, particularly if you want to avoid repeating one of those long, heavily qualified object names. For example, if you have this heavily qualified string object and you want to add the characters *ED* to it, in traditional VB you must type the object's name twice, like this:

```
MyAssem.MyNameSpace.MyObject = MyAssem.MyNameSpace.MyObject +
    "ED"
```

But in VB .NET, you can omit the second object name, thereby bringing the = and the + together into +=, like this:

```
MyAssem.MyNameSpace.MyObject += "ED"
```

It takes a little getting used to, but it's an improvement. The fundamental difference is that the C-style moves the operator (+, -, *, or whatever) way over next to the assignment (=) symbol.

So you get:

```
X += 1 ' (The new C-style syntax)
```

instead of:

```
X = X + 1
```

If you want to try out the C syntax, here are the variations:

<code>X = X + Y</code>	<code>X +=Y</code>
<code>X = X - 5</code>	<code>X -= 5</code>
<code>X = X * 4</code>	<code>X *= 4</code>
<code>X = X / 7</code>	<code>X /= 7</code>
<code>X = X ^ 2</code>	<code>X ^= 2</code>
<code>String1 = String1 + "ing"</code>	<code>String1 += "ing" (you can also use &=)</code>

Overloaded Functions, Properties, and Methods (Overloading)

Before VB .NET

Sometimes you may want to use the same function name but accomplish different tasks with it. Typically, you would do this by using optional parameters:

```
MyFunction (Optional SomeParam As Variant)
    If IsMissing(SomeParam) Then
        'Do one task because they didn't send SomeParam
    Else
        'Do a different task because they did send SomeParam
    End If
```

This way, the same `MyFunction` behaves differently based on what you pass to it.

You can still use optional parameters, but their utility has been reduced in VB .NET. Among other things, the `IsMissing` command is no longer part of the VB language. This puts a slight crimp in things if you attempt to write code like the previous example. For details about what has been removed or restricted concerning optional parameters, see “Optional Parameter-Passing Tightened” in this appendix.

VB .NET

You can use the same function name to do different tasks in VB .NET, but you employ a different technique than using the `Optional` command. (Properties and methods can also be overloaded.)

In VB .NET, you can overload a function by writing several versions of that function. Each different version of the function uses the same function name but has a different argument list. The differences in the argument list can be different data types, or a different order, or a different number — or two or three of these variations at once.

The VB .NET compiler can tell which version of this function should be used by examining the arguments passed to the function (the order of the arguments, the number of arguments, and/or the different variable types of the arguments).

VB .NET employs overloading quite extensively in the language itself. You can see it whenever VB .NET lists various different argument lists you can choose from. The `MessageBox.Show` method, for example, is really overloaded: It will recognize 12 different kinds of argument lists. Try this. Type the following line into the code window within the `Form_Load` event:

```
messagebox.Show(
```

As soon as you type that left parenthesis, VB .NET pops up a gray box showing the first of the 12 different ways you can use the `MessageBox` object's `Show` method. The box says: 1 of 12. Use your keyboard's up- and down-arrow keys to see all the different ways that this overloaded method can be used.

Doing it yourself

Why should the people who wrote the VB .NET source code have all the fun? Here's an example of how you, too, can use overloading in your own VB .NET source code when writing your own applications.

Let's say that you want a function named `SearchBooks` that will give you a list of a particular author's book titles if you only provide the author's name. However, at other times you want `SearchBooks` to behave differently: You want a list limited to the author's books in a particular year if you also provide the year by passing that parameter to the function.

Create two overloaded functions with the same name, which accept different arguments:

```
Public Overloads Function BookList(ByVal AuthorName As
    String) As Object

    ' write code to find all books by this author

End Function

Public Overloads Function BookList(ByVal AuthorName As
    String, ByVal YearNumber As Integer) As Object

    ' write code to find all books written in the particular year

End Function
```

Some programmers like to use the overload feature to group a set of identically named functions that, as this example illustrates, handle different jobs. To me, this has the potential of making the source code harder to read and harder to maintain. If you want to accomplish two jobs, you can write two different functions — and give them descriptive names so that you can tell what they do when you read the source code:

```
Public Function GetBooksByYear (ByVal AuthorName As String,
    ByVal YearNumber As Integer) As String

Public Function GetAllBooksOfAuthor (ByVal AuthorName As
    String) As String
```

However, overloading can be of value to many programmers. VB .NET itself uses overloading extensively, as you can easily see by typing a line like this in the code window:

```
console.WriteLine(
```

As soon as you type the parenthesis symbol, a small window opens with scrollable arrow symbols and displays 2 of 18, meaning that you are currently looking at the second of 18 different overloaded versions of the `WriteLine` function (method). Press your up- or down-arrow keys to scroll through all 18 different argument lists for this `WriteLine` procedure.

.NET creates unique signatures for overloaded functions. These signatures are based on the name of the function and the number and types of the parameters.



As a side note, the signature is not affected by type modifiers (`Shared` or `Private`), parameter modifiers (`ByVal` or `ByRef`), the actual names of the parameters, the return type of the function, or the element type of a property. A function with three parameters, two of which are optional, will generate three signatures: one signature with all three parameters, one signature with the required parameter and one optional parameter, and one signature with just the required parameter.

Overloading is also used as a replacement for the `As Any` command. `As Any` has been deleted from the language, but you can use overloading to permit functions to return several different data types. However, you can no longer declare a function `As Any` (meaning it can return any kind of data type):

```
Public MyFunction (parameters) As Any
```

When you declare a function, every parameter and the function itself (the return type) must be specified.

As Any is gone

In Visual Basic 6.0, you could specify `As Any` for the data type of any of the parameters of a function, and for its return type. The `As Any` keywords disable the compiler's type checking and allow variables of any data type to be passed in or returned. The `As Any` command has been removed in VB .NET because `As Any` degrades type safety. If you want to permit more than one return type for a procedure, use overloading.

Parameters (If Not Needed)

Before VB .NET

When passing traditional parameters to a function, you can omit parameters that you don't want to specify by simply leaving out their names. For example, if you wanted to specify the first and third parameters for a `MsgBox`, you could just leave out the second parameter, like this:

```
MsgBox("Your Name", , "Title")
```

VB understood that the empty area between commas in a parameter list meant that you wanted to use the default, or no parameter at all.

VB .NET

That tactic still works with many functions and methods in VB .NET, but not always!

Sometimes, you must use the word `Nothing` instead.

For example, when you want to sort a `DataRow`, the first parameter specifies a filter (such as "all dates after 12/12/01"). (The second parameter specifies the column to sort on, so you do want to specify that parameter.)

However, if you want to avoid using a filter — if you want all the rows sorted, for example — you must use `Nothing` in the parameter list. Leaving the first parameter out raises an error message:

```
Dim fx As String = "title"  
'dt is a DataTable in a DataSet  
myRows = dt.Select(, fx)
```

Instead, you must use `Nothing`:

```
Dim fx As String = "title"  
myRows = dt.Select(Nothing, fx)
```

Parent Property

This new property specifies the parent container of this control.

Parentheses Now Required

Before VB .NET

The `Call` command is ancient. It has been used since the first versions of Basic more than 20 years ago. It indicates that you're calling a procedure.

`Call` is optional, but traditional VB has required that if you use it, you must enclose passed parameters within parentheses:

```
Call MySub (N)
```

But if you omitted the `Call` command, you did not use parentheses:

```
MySub N
```

You could also add extra parentheses, which had the same effect as using the `ByVal` command (the called procedure cannot then change any passed parameters):

```
Call MySub ((N))  
MySub (N)
```

VB .NET

VB .NET requires parentheses in all cases (the `Call` command remains optional):

```
Call MySub (N)  
MySub (N)
```

To translate VB 6 and earlier code to VB .NET, you must go through and add parentheses to any procedure call that doesn't employ them (that does not use the `Call` command).

Point Changed

See "Circle, Line, Shape, PSet, and Point All Changed."

Print Is No Longer Available

The `Print` command can no longer be used with a form. In other words, the following code doesn't work:

```
Print 12 * 3
```

In earlier versions of VB, if you put that code in your program, the number 36 would appear on the current form. No more.

VB 6 (and earlier) programmers often used this `Print` command to quickly test or debug their programs. It was a superior alternative to the `Debug.Print` command (which requires that you look for the results in a special Immediate window). The `Print` command put the results right there on your form as soon as you pressed F5.

This feature has been removed in VB .NET.

Printing to a Printer Has Changed

Before VB .NET

To print the contents of a `TextBox` to the printer, you typed the following:

```
Printer.Print Text1
```

To print a string, you typed the following:

```
Printer.Print "This."
```

To print a whole form (its graphic appearance), you typed the following:

```
PrintForm
```

VB .NET

In VB .NET, you have greater control over what gets printed, but there is a cost. Some would say a terrible cost. You use a group of MFC (an API used by C programmers) members. You have to muster a fair amount of information (such as “brush” color, the height of each line of text, and so on), and you have to manage several other aspects of the printing process as well.

Project Properties

If you want to change the name of the assembly, change which object (such as Form2) is supposed to be the startup object (the one that executes first when the project is run), or change other fundamental aspects of your project, you cannot do it from the menu system anymore. In VB .NET, you must right-click the name of your project, and then choose Properties.

Property Definitions Collapse (Property Let, Get, and Set Are Gone)

Before VB .NET

To expose a property of an object, you used the `Property Get`, `Property Set`, and `Property Let` procedures. Each of these was a separate, discrete procedure. Therefore, you could freely specify varying scopes for the different procedures. For example, you could define the `Get` (read) procedure as `Public`, but refuse to permit outsiders to change the property simply by making the `Let` (write) procedure `Private`. (The `Friend` scoping definition was particularly useful in this regard. You could use it to give your procedures in your control access to each other’s properties but block any code outside your control from having access to certain elements that you want to protect.)

VB .NET

Now in VB .NET, you must combine `Get` and `Set` within a single logical block of code. This has the effect of allowing you to define scope for the property as a whole (you cannot specify different scopes for `Get` and `Set`).

Whither Let?

Everything in VB .NET is an object, so the `Let` command disappears. You can only use `Get` and `Set`. Here's how you must now code a property:

```
Private m_MyProperty As String

Public Property MyProperty As String
Get
    MyProperty = m_MyProperty
End Get

Set
    M_MyProperty = MyProperty
End Set

End Property
```

And don't you dare suggest that this source code seems a little bit redundant, or that VB .NET could automatically fill in all this repetition for you, so that you would have to write only `Public Property MyProperty As String` and all the other stuff would be automatically inserted into the code window for you. It's true, but don't suggest it. Fortunately for us, somebody finally did suggest it, and VB .NET 2003, the latest version of the language, does fill in this entire property declaration template.

Specifying different scope levels

If you want different scope levels, you have to define two different properties but have them manipulate the same internal variables. This is what we used to call a workaround. Here's an example that makes the read (`Get`) public, but restricts the write (`Set`) to `Friend` (control-internal only):

```
Private m_MyProperty As String

ReadOnly Public Property MyProperty As String
Get
    MyProperty = m_MyProperty
End Get
End Property

WriteOnly Friend Property SetMyProperty As String

Set
    M_MyProperty = MyProperty
End Set
End Property
```

PSet Changed

See “Circle, Line, Shape, PSet, and Point All Changed.”

Public versus Property Procedures

Before VB .NET

Any changes made to a `Public` variable in a class are not seen by the calling procedure. For example, changes made by `AFunction` to `TheObj.ItsVariable` in the following code will not be seen by the calling procedure:

```
AFunction(TheObj.ItsVariable)
```

VB parses this function call and merely passes a temporary variable to `AFunction`. That temporary variable is not passed back to the calling procedure, so any modifications made to the variable are then lost.

If you want changes to a `Public` variable to persist in the calling procedure, you must first explicitly assign that variable’s value to a different variable. This is how you could permit a called function to modify an object’s property value (without using a `Property` procedure):

```
NewVariable = TheObj.ItsVariable  
AFunction(NewVariable)  
TheObj.ItsVariable = NewVariable
```

VB .NET

The workaround described in the previous code is unnecessary in VB .NET. `Public` variables within a class can be directly modified by a called function (in effect, a `Public` variable can be considered global to the project). Using the previous example, `AFunction` can change `ItsVariable` if you send simply and directly like this:

```
AFunction(TheObj.ItsVariable)
```

Permitting outside procedures to have direct access to `Public` variables in a class is something that you should probably avoid. I suggest you stick to using traditional property procedures to manage an object’s properties and avoid this new VB .NET feature that exposes `Public` variables.

Reading the Registry

Before VB .NET

In VB 6 and before, you could use API commands such as `RegQueryValueEx` to query the Registry. Or you could employ the native VB Registry-related commands, such as `GetSetting`, like this:

```
Print GetSetting(appname := "MyProgram", section := "Init",  
                key := "Locale", default := "1")
```

VB .NET

In VB .NET, you can query the Registry using the `RegistryKey` object. However, you're not really supposed to use the Registry much in VB .NET. The idea of avoiding the Registry is that in VB .NET, you are supposed to steer clear of the traditional DLL problems (which version should be loaded if two applications use two different versions of a particular "dynamic link library?"). These problems have plagued us for many years.

In VB .NET, you put all support files (including code libraries, now called assemblies) in the same directory path as the VB .NET application. In other words, you put all the files needed by your application into its directory (or a subdirectory under that directory). That way, you can "photocopy" (make an X-copy, or directly just copy the VB .NET application's directory and all its subdirectories), and by simply doing this copy, you deploy (transfer to another computer) your VB .NET application. No more need for Setup programs to install your application; no more need to modify the Windows Registry. (Of course, all this assumes that the huge VB .NET runtime library — the CLR as it's called — is already installed on the other computer to which you are deploying your VB .NET application. At some point, Microsoft says, nearly everyone will have the CLR because it will be built into the operating system (Windows) or the browser (Internet Explorer) or otherwise made universally available.

Where, though, should a VB .NET programmer store passwords or other customization information (such as the user's choice of default font size) instead of the Registry that you've used for the past several years? What goes around comes around. Go back to using a good old .INI file or a similar simple text file. It's quick and easy, and it avoids messing with the Registry.

Using the Registry

If you must use the Registry, though, here's how to access it from VB .NET. Start a new VB .NET WinForm-style project and then double-click `TextBox` in the WinForms tab of the Toolbox to add it to your `Form1.VB`. Double-click a Button to add it as well.

Click the `TextBox` in the IDE (Design tab selected) so that you can see its properties in the Properties window. Change its `Multiline` property to `True`.

Now, double-click the Button to get to its Click event in the code window. Type this into the Button's Click event:

```
Private Sub Button1_Click_1(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles
    Button1.Click

    Dim objGotValue As Object
    Dim objMainKey As RegistryKey = Registry.CurrentUser
    Dim objOpenedKey As RegistryKey
    Dim strValue As String

    objOpenedKey =
        objMainKey.OpenSubKey("Software\Microsoft\Window
        s\CurrentVersion\Internet Settings")

    objGotValue = objOpenedKey.GetValue("User Agent")

    If (Not objGotValue Is Nothing) Then
        strValue = objGotValue.ToString()
    Else
        strValue = ""
    End If

    objMainKey.Close()

    TextBox1.Text = strValue

End Sub
```

You must also add `Imports Microsoft.Win32` up there at the top of the code window where all those other `Imports` are. The `Microsoft.Win32` namespace contains the Registry-access functions, such as the `OpenSubKey` method, that you need in this example.

Press F5 to run this example and click the Button. If your Registry contains the same value for this key as my Registry contains, you should see a result similar to this:

```
Mozilla/4.2 (compatible; MSIE 5.0; Win32)
```

Note that the complete name (path) of the entire Registry entry is divided into three different locations in the example code (they are in boldface): first the primary key, `CurrentUser`, then the path of sub-keys, and finally the actual specific “name”: `objOpenedKey.GetValue("User Agent")`.

Writing to the Registry

The `RegistryKey` class includes a group of methods you can use to manage and write to the Registry. These methods include `Close`, `CreateSubKey`, `DeleteSubKey`, `DeleteSubKeyTree`, `DeleteValue`, `GetSubKeyNames`, `GetType`, `GetValue`, `GetValueNames`, `OpenSubKey`, and `SetValue`.

ReDim

Before VB .NET

You can use the `ReDim` command to initially declare a dynamic array. Simply using `ReDim` rather than `Dim` makes the array dynamic (changeable elsewhere in the code), and also declares (and dimensions) the array, just as `Dim` does.

VB .NET

All VB .NET arrays can be redimensioned, so there is no need for a special subset of “dynamic” arrays. `ReDim` can no longer be used as an alternative to the `Dim` command. Arrays must be first declared using `Dim` (or similar) — because `ReDim` cannot create an array. `ReDim` can be used only to redimension (resize) an existing array originally declared with `Dim` (or similar) command.

Also see “Dim and ReDim: New Rules.”

References to Code Libraries

See “Imports.”

Region Property

This new property specifies the region (an area within a window) that is used with this control.

ResizeRedraw Property

This new property specifies whether the control should be redrawn when it is resized. In some cases, a graphic or text on a resized control will need to be redrawn to appear correctly.

Return

Before VB .NET

To return a value from a function, you assigned that value to the function's name:

```
Function AddFive(ByVal N As Integer) As Integer
AddFive = N + 5
End Function
```

VB .NET

You now have the option of using the more readable `Return` command:

```
Function AddFive(ByVal N As Integer) As Integer
Return N + 5
End Function
```

Right Property

This new property tells you the distance between the right edge of the control and the left edge of its container.

Right (RightStr or Right\$) Must Be “Qualified”

Before VB .NET

To access characters on the right side of a string, you used the `Right` function, like this:

```
If Right(filepathname, 1) <> "\" Then filepathname =  
    filepathname & "\"
```

VB .NET

The Form “object” now has a `Right` property, which means that if you want to use the `Right` function, you must add some qualifiers (adjectives describing the class to which this version of “right” belongs):

```
If Microsoft.VisualBasic.Right(filepathname, 1) <> "\" Then  
    filepathname = filepathname & "\"
```

The same requirement applies if you want to use the `Left` function. VB 6 could tell the difference between these uses of the words `Right` and `Left` by looking at the context in the source code and distinguishing between a property and a function. In VB .NET, the burden of making this distinction shifts to the programmer.

When you use the qualifier `Microsoft.VisualBasic`, you are invoking the “compatibility” namespace, which provides for some backward compatibility with VB 6 programming code.

You can also use this compatibility namespace if you want to continue using the various traditional string-manipulation functions such as `Mid`, `Instr`, and so on. However, it is better to become familiar with all the new and more efficient .NET string functions and methods. See “String Functions Are Now Methods” in this appendix for details and the VB .NET equivalents of traditional techniques.

RND Has New Syntax (It's an Object Now)

Before VB .NET

In VB 6 and previous versions, you would generate a random number between 1 and 12 like this:

```
X = Int(Rnd * 12 + 1)
```

Or to get a random number between 0 and 12, you would use this code:

```
X = Int(Rnd * 13)
```

You used the `Rnd` and `Randomize` functions.

VB .NET

It's different now. You must use a `System.Random` object. The good news is that the `Random` object has useful capabilities that were not available via the old `Rnd` and `Randomize` functions.

Type this code into a form's Load event:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e  
    As System.EventArgs) Handles MyBase.Load  
    Dim i As Integer  
    For i = 1 To 100  
        Debug.Write(rand(i) & " ")  
    Next  
End Sub
```

And elsewhere in that form's code window, type this function, which returns random numbers between 1 and 12:

```
Function rand(ByVal MySeed As Integer) As Integer  
    Dim obj As New System.Random(MySeed)  
    Return obj.Next(1, 12)  
End Function
```

When you press F5 to run this example, you see the `Debug.Write` results in the output window in the IDE.

Although the arguments say 1, 12 in the line `Return obj.next(1, 12)`, you will not get any 12s in your results. The numbers provided by the `System.Random` function in this case will range only from 1 to 11. I'm hoping that this error will be fixed at some point. However, even the latest version of VB .NET (2003) gives you results ranging only from 1 to 11. I'm hoping even more strongly that this truly odd behavior is not the result of an intentional "feature" of the `Random.Next` method. It's just too tedious to hear justifications for confusing programmers with upper boundaries like this 12 that turn out — surprise! — not to be upper boundaries after all.

On a happier note, here's an example that illustrates how you can use the `Now` command to seed your random generator. Put a Button control on your form and then type in the following:

```
Private Sub Button1_Click_1(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button1.Click

    Dim sro As New coin()
    Dim x As Integer
    Dim i As Integer

    For i = 1 To 100
        sro.toss()

        Dim n As String

        x = sro.coinvalue
        If x = 1 Then
            n = "tails"
        Else
            n = "heads"
        End If

        n = n & " "

        debug.Write(n)

    Next i

End Sub

End Class

Class coin

    Private m_coinValue As Integer = 0

    Private Shared s_rndGenerator As New
        System.Random(Now.Millisecond)
```

```
Public ReadOnly Property coinValue() As Integer
    Get
        Return m_coinValue
    End Get
End Property

Public Sub toss()
    m_coinValue = s_rndGenerator.next(1, 3)
End Sub

End Class
```

As always in VB .NET, there is more than one way to do things. The next example uses the `System.Random` object's `Next` and `NextDouble` methods. The seed is automatically taken from the computer's date/time (no need to supply a seed as was done in the previous example, or to use the old VB 6 `Randomize` function). The next method returns a 32-bit integer; the `NextDouble` method returns a double-precision floating point number ranging from 0 up to (but not including) 1.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load
    Dim r As New System.Random()
    Dim x As Integer
    Dim i As Integer

    For i = 1 To 10
        Debug.Write(r.Next & ", ")
    Next

    For i = 1 To 10
        Debug.Write(r.NextDouble & ", ")
    Next

End Sub
```

The following is a sample of what you see in the output window when you run this example:

```
519421314, 2100190320, 2103377645, 526310073, 1382420323,
408464378, 985605837, 265367659, 665654900,
1873826712
0.263233027543515, 0.344213118471304, 0.0800133510865333,
0.902158257040269, 0.735719954937566,
0.283918539194352, 0.946819610403301,
0.27740475408612, 0.970956700374818,
0.803866669909966
```

Any program that uses this technique can be guaranteed to get unique results each time it's executed. It's impossible to execute that program twice at the same time and date, just as you cannot kiss yourself on the face. (Even Mick Jagger can't.)

Of course, if you choose, you can still employ the older VB 6 version of the randomizing function:

```
Dim r As Double
    r = Rnd
    MsgBox(r)
```

It's not necessary in this case to invoke the `Microsoft.VisualBasic` namespace. The `Rnd` function is a wrapper, like many other attempts at backward compatibility between VB .NET and earlier VB code (such as `MsgBox` rather than the new `MessageBox.Show`).

If you've used the `Rnd` function before, you may recall that it will provide identical lists of random numbers by default (which can be quite useful when attempting to, for example, debug a game). In fact, I use this feature in my unbreakable code system described in the Warning in the next section. If you want to get identical lists in VB .NET, you can seed the `Random` object, like this:

```
Dim r As New System.Random(14)
```

Filling an array

The `Random.NextBytes` method automatically fills an array of bytes with random numbers between 0 and 255. Here's how:

```
Dim r As New System.Random()
Dim i
Dim a(52) As Byte 'create the byte array

r.NextBytes(a) ' fill the array

For i = 0 To a.Length - 1
    Debug.WriteLine(i.ToString + ". " + a(i).ToString)
Next
```



Neither the old `Rnd` function nor the new `Random` object uses algorithms that are sufficiently sophisticated to be of direct use in most kinds of cryptography. Nonetheless, an unbreakable encryption system can be built upon them. (I describe and include an application that does this in my book *Hacker Attack*, published by Sybex.) However, if you want to explore encryption in VB .NET, you would be well advised to check out the `System.Security.Cryptography` namespace and the objects it offers.

RTrim

See “LCase and VCase Change.”

ScaleHeight and ScaleWidth Are Gone

Before VB .NET

You could get a measurement of the inside dimensions of a form (ignoring its frame) by using this code:

```
X = ScaleWidth  
Y = ScaleHeight
```

VB .NET

Now you must use the following:

```
Dim X as Integer, Y as Integer  
  
X = ClientSize.Width  
Y = ClientSize.Height
```

The results, however, are provided only in pixels.

Scope for Modules

If you want to make some variables (or enums) visible throughout an assembly or throughout a “solution,” put them in a module, but use `Public Module` rather than simply `Module`. Modules without the `Public` declaration are visible only within the project in which the module resides.

SelStart and SelLength Are Renamed (Selected Text)

Before VB .NET

You identified text that the user selected in a `TextBox`, like this:

```
Text1.SelStart = WhereLoc - 1 ' set selection start and  
Text1.SelLength = Len(SearchString) ' set selection length
```

VB .NET

If you want to get a copy of selected text in a `TextBox` in VB .NET, use this code:

```
Dim s As String
s = TextBox1.SelectedText
```

You must go through a rather dramatic OOP acrobatic performance when you use the `ActiveControl` method. Also, `SelLength` becomes `SelectedLength`, `SelStart` becomes `SelectedStart`, and so on.

SendKeys Changes

You can use the `Shell` command to run an application, such as Notepad. `Shell` works much as it did in VB 6. An associated command, `SendKeys`, imitates the user typing on the keyboard. `SendKeys` works differently in VB .NET. This code will run an instance of Windows Notepad, and then “type” This Message into Notepad:

```
Dim X As Object
X = Shell("notepad.exe", AppWinStyle.NormalFocus)
System.Windows.Forms.SendKeys.Send("This Message")
```



If you put this code in a `Form_Load` event, it will only send the `T` into Notepad (there are timing problems involved). So, put it into a different event, such as `Button1_Click`, and VB .NET will have enough time to get itself together and send the full message.

Set Is No More

Remove all uses of the `Set` command. Just write the same line of code as before, but without `Set`.

Before VB .NET

Here’s an example of how the `Set` command was used:

```
Set MyDataConn = Server.CreateObject("ADODB.Connection")
```

To accomplish the preceding job in VB .NET, leave out the `Set` command.

Also note that if you tried to change an object variable so that it referenced a different object before VB .NET, you used this syntax:

```
Set CurrentObjVar = DifferentObjVar
```

Before VB .NET, you could not leave out the `Set` command, like this:

```
CurrentObjVar = DifferentObjVar
```

If you did, something entirely different occurred. If the `DifferentObjVar` had no default property, an error was triggered. If it did have a default property, that default property was assigned to `CurrentObjVar` as its default property.

VB .NET

VB .NET doesn't have the `Set` command, and in general it doesn't allow default properties anyway. For example, you can no longer use `x = Text1` as a way of assigning the text in a `TextBox` to the variable `x`. Instead, you must explicitly name any property, including what were previously "default" properties. In VB .NET, this works like so: `x = Textbox1.Text`.

However, just to make life interesting, some objects do have default properties that you can use in VB .NET. Specifically, objects that take arguments (the dictionary object, the `Item` property in a collection, parameterized properties, and so on) can be assumed to be defaults and explicit reference to the property can, in these few cases, be omitted from your source code.

Parameterized properties (properties that take parameters) are still permitted to be defaults in VB .NET. For example, in VB .NET, most collections fall into this category. Here are some examples from the ADO data language:

```
Recordset object    (its default property is Fields)
Fields collection  (its default property is Item)
Field object       (its default property is Value)
```

For instance, this code illustrates two ways to reference the `Item` property of a recordset's `Field` object:

```
Dim rs As Recordset
```

You can use the fully qualified code:

```
rs.Fields.Item(1).Value = "Hello"
```

Alternatively, you can omit the term `Item` because it is the default property of the `Fields` collection:

```
rs.Fields(1).Value = "Hello"
```

For more information on instantiating objects in VB .NET, see “Object Instantiation” in this appendix.

SetFocus

See “Focus.”

Shape Control Is Gone

See “Circle, Line, Shape, PSet, and Point All Changed.”

ShowFocusCues Property

This new property tells you if the form will currently display one of those visual cues (like the dotted gray rectangle inside a Button control) that indicates which control has the focus.

ShowKeyboardCues Property

This new property tells you whether the form will currently display those keyboard shortcuts (underlined letters in the `Text` property of controls).

Size Property

This new property specifies the height and width of the control (in the format `width, height`).

Sorting a DataTable

See “Parameters (If Not Needed)”

Static Procedures Are No Longer Available

Before VB .NET

Normally, if you use a variable only within a procedure, it lives and dies within that procedure. Its scope is said to be local — so no other procedures can read or change that variable. However, the value held in a local variable also evaporates when the procedure that contains it is no longer executing. Sometimes you want a variable to be local in scope, but you want its value to persist. (Another way to put this is that you sometimes want a variable to be visible only within its procedure, but you want it to have a lifetime greater than merely procedure-level.)

In traditional VB, you can cause a local variable's value to persist by using the `Static` command to declare it:

```
MySub DoSomething
  Static X
  X = X + 1
End Sub
```

Each time this procedure executes, the value in variable `x` is increased by 1. However, if you omit the `Static` keyword, `x` reset to 0 whenever the procedure is finished executing; therefore `x` simply changes from 0 to 1 each time the procedure executes.

You could also declare all the variables in a procedure to be static at once by merely defining the entire procedure as static:

```
Static MyFunction()
```

This procedure use of `Static` is not supported in VB .NET.

VB .NET

The `Static` command can be used when declaring individual variables.

However, you cannot declare an entire procedure as `Static` in VB .NET.

String Functions Are Now Methods

A string used to be a variable type; now it's an object, and it has many methods.

Where previous versions of VB used functions, VB .NET often uses methods — behaviors that are built into objects. In VB .NET, everything is an object, even a string variable. So, you shouldn't be surprised that to find the length of a string in VB 6, you used the function `Len(String)`, but now in VB .NET, you use the method `String.Length`. Fortunately, the old functions such as `Len` still work in VB .NET, so usually the approach you choose — function or method — is up to you.

In addition to those described in the preceding sections, you might want to check out VB .NET Help for information on how to use these new string methods, just so you know that they exist if you ever need them: `Compare`, `Concat`, `Format`, `Chars`, `EndsWith`, `Insert`, `LastIndexOf`, `PadLeft`, `PadRight`, `Remove`, `Replace`, `Split`, and `StartsWith`. There are others, such as `Trim`, that are discussed in the upcoming sections.

The various string methods or functions can manipulate text in many ways. Some of them take more than one argument. Arguments can include string variables, string literals, or expressions.

IndexOf or InStr

The traditional VB `InStr` format is as follows:

```
InStr([start, ]string1, string2[, compare])
```

`InStr` tells you where (in which character position) `string2` is located within `string1`. Optionally, you can specify a starting character position and a comparison specification that you can ignore — the default is what you want for the comparison style.

This is a remarkably handy function when you need to parse some text (meaning that you need to locate or extract a piece of text within a larger body of text). `InStr` can enable you to see if a particular word, for example, exists within a file or within some text that the user has typed into a `TextBox`. Perhaps you need to search a `TextBox` to see if the user typed in the words `New Jersey`, and if so, to tell him or her that your product is not available in that state.

`InStr` is case-sensitive by default; it makes a distinction between Upper and upper, for example.

What if you want to know whether more than one instance of the search string is within the larger text? You can easily find additional instances by using the result of a previous `InStr` search. `InStr`, when it finds a match, reports the location and the character position within the larger text where the search string was found, as in the following example:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    Dim quot, MainText, SearchWord As String
    Dim X, Y, Z As Integer

    quot = Chr(34)

    MainText = "Masterpieces are built of pieces."
    SearchWord = "pieces"

    Do
        X = Y + 1
        Z = Z + 1
        Y = InStr(X, MainText, SearchWord)

    Loop Until Y = 0

    MsgBox("We found " & SearchWord & " " & Z - 1 & _
        " times inside " & quot & MainText & quot)

End Sub
```

In this example, the loop continues to look through the `MainText` until the `InStr` function returns a zero (which indicates that the `SearchWord` isn't found any more). The variable `Z` is used to count the number of successful hits. The variable `X` moves the pointer one character further into the `MainText` (`X = Y + 1`). You can use this example as a template any time you need to count the number of occurrences of a string within another, larger string.

To merely see if, in the previous code, a string appears within another one, use this:

```
If InStr("Masterpiece", "piece") Then MsgBox "Yep!"
```

The preceding code line translates to: If "piece" is found within "Masterpiece," then display "Yep!"

There's also an `InStrRev` function that works in a similar way, but it starts looking at the last character and searches backward to the first character.

The equivalent VB .NET method is `IndexOf`. Here's an example that finds the first occurrence of the letter *n* in a string:

```
Dim s As String = "Hello Sonny"
Dim x As Integer
x = s.IndexOf("n")
MsgBox(x)
```

`IndexOf` is case-sensitive. To specify the starting character position, add an integer to the argument list, like this:

```
x = s.IndexOf("n", x)
```

To translate the Masterpiece example, change these two lines:

```
Y = MainText.IndexOf(SearchWord, X)
Loop Until Y = -1
```

ToLower or LCase (String)

`LCase` removes any uppercase letters from a string, reducing it to all lower-case characters. *AfterWord* becomes *afterword*. Likewise, there's also a `UCase` function that raises all the characters in a string to uppercase.

The VB .NET `ToLower` method can be used instead of `LCase`, and VB .NET `ToUpper` replaces `UCase`.

These functions or methods are used when you want to ignore the case — when you want to be case-insensitive. Usually, `LCase` or `UCase` is valuable when the user is providing input, and you cannot know (and don't care) how he or she might capitalize the input. Comparisons are case-sensitive:

```
If "Larry" = "larry" Then MsgBox "They are the same."
```

This `MsgBox` will never be displayed. The *L* is not the same. You can see the problem. You often just don't care how the user typed in the capitalization. If you don't care, use `LCase` or `UCase` to force all the characters to be lower-case or uppercase, like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    Dim reply As String
    Dim n As Integer

    reply = InputBox("Shall we proceed?")
```

```
reply = UCase(reply)

Dim x As Integer

If reply = "YES" Then
    MsgBox("Ok. We'll proceed.")
End If

End Sub
```

Notice that it now does not matter how the user capitalized *yes*. Any capitalization will be forced into uppercase letters, and you compare it to a literal that is also all uppercase.

To translate this to VB .NET, change the following line:

```
reply = UCase(reply)
```

to this:

```
reply = reply.ToUpper
```

Substring or Left (String, Number)

The `Left` function returns a portion of the string, the number of characters defined by the `Number` argument. Here's an example:

```
Dim n As String

n = Microsoft.VisualBasic.Left ("More to the point.", 4)
MsgBox(n)
```

This code results in `More`.

There's also a `Right` function. Both `Left` and `Right` require the `Microsoft.VisualBasic` qualifier, which was not necessary in previous versions of VB.

The VB .NET equivalent of `Left` or `Right` is the `SubString` method:

```
Dim n As String = "More to the point."

n = n.Substring(0, 4)
MsgBox(n)
```

Or to get a string from the right side, the following code retrieves all characters from the 12th character to the end of the string:

```
n = n.Substring(12)
```

Length or Len (String)

This function tells you how many characters are in a string. You may want to let users know that their response is too wordy for your database's allotted space for a particular entry. Or perhaps you want to see if they entered the full telephone number, including their area code. If they did, the number will have to be at least 10 characters long. You can use the less-than symbol (<) to test their entry, like this:

```
If Len(TextBox1.Text) < 10 Then  
    MsgBox("Shorter")  
End If
```

The VB .NET equivalent is the `Length` method:

```
If TextBox1.Text.Length < 10 Then
```

The Trim Method or LTrim (String)

`LTrim` (and its brother `RTrim`) removes any leading (or trailing) space characters from a string. The uses for this function are similar to those for `UCase` or `LCase`: Sometimes people accidentally add extra spaces at the beginning or end of their typing. Those space characters will cause a comparison to fail because computers can be quite literal. " This" is not the same thing as "This". And if you write the code `If " This" = "This"`, and the user types in " This ", the computer's answer will be no.

The VB .NET equivalent is the `Trim` method. Here's an example that removes four leading spaces:

```
Dim s As String = "    Here"  
s = s.Trim  
MsgBox(s & s.Length)
```

Substring or Mid

The `Mid` format is:

```
Mid(String, StartCharacter [, NumberOfCharacters])
```

You find yourself using this function or method surprisingly often. It's a little like the `Left` and `Right` functions, except `Mid` can extract a substring (a string within a string) from anywhere within a string. `Left` and `Right` are limited to getting characters on one end of a string or the other. The VB .NET `Substring` method does all of these things.

The `Mid` function works like this:

```
MsgBox(Mid("1234567", 3, 4))
```

This results in 3456. You asked `Mid` to begin at the third character and extract four characters, and that's exactly what you got.

The VB .NET `Substring` equivalent is:

```
Dim s As String = "1234567"  
MsgBox(s.Substring(2, 4))
```

Notice that to start with the third character in this string (the digit 3), you must specify 2 in your `Substring` argument. Absurd isn't it? This nonsense is because the `Substring` method begins counting characters with 0. Somebody who worked on designing VB .NET thinks we should start counting with 0 instead of 1.

The VB 6 `Mid` function begins counting characters with 1, as we humans do. In this case, VB .NET takes a step backward by counting from 0 in the `Substring` method.

The StartsWith Method

This method gives you a quick way of telling whether or not a string begins with a particular character or substring:

```
Dim s As String = "narop"  
If s.StartsWith("na") Then MessageBox.Show("Yes")
```

String Manipulation Using the StringBuilder

Before VB .NET

The string was simply a variable type.

To display a large amount of text in Visual Basic, it's sometimes necessary to resort to a process that builds strings. Rather than assigning a paragraph-long piece of text to a string variable, you instead use the `&` operator to build a large string out of smaller strings, like this:

```
StrLabelText = "Thank you for submitting your information."  
StrLabelText = StrLabelText & "We appreciate your input. "  
StrLabelText = StrLabelText & "If all our customers were as  
responsive and submissive "
```

Then you assign the string to a label or display it in some other way to the user:

```
LblResponse.Text = strLabelText
```

VB .NET

The string, like most everything else, is an object, and it has many methods. VB .NET also offers you a more efficient (faster executing) way of manipulating strings, called the `StringBuilder`. Use it when you are going to do some heavy-duty manipulations of a string.

Instead of creating a new string each time you modify a string, the `StringBuilder` does not spawn multiple string objects (as do ordinary VB .NET string methods, such as `ToUpper`). Why? In VB .NET, when you create a string, it is immutable — meaning it cannot be changed. If you ask for a change (like “make this string all uppercase” with the `ToUpper` method), a brand new string is created with all uppercase letters, but the old string hangs around until garbage collection cleans things up. Asking VB .NET to create new strings each time you manipulate a string wastes time and space. (The `StringBuilder` is not efficient when you are only reading or querying the contents of a string — such as `IndexOf`. The `StringBuilder` is useful when you are changing a string, not merely reading it.)

If you are writing some source code involving repeatedly changing a string, you can speed up your program's execution by creating a `StringBuilder` object rather than using normal strings. After you are through with the manipulations, you can turn the `StringBuilder`'s products back into normal strings. The `StringBuilder` sets aside some memory for itself when you instantiate it — then it does its work directly on the string data within that memory block. No nasty little abandoned strings hanging around waiting for the garbage truck to make its rounds.

Also the `StringBuilder` is extremely rich with features. You'll want to use it.

To create a `StringBuilder`, you first `Imports` the `System.Text` namespace up at the top of your code window. In a `WebForm`, for example, you use this code:

```
<%@ Import Namespace="System.Text" %>
```

or in a WinForm, type this:

```
Imports System.Text
```

Then you declare a new `StringBuilder`. The `StringBuilder` offers six different constructors (read about this in “Constructors Are Now ‘Parametized’” in this appendix). In other words, when you declare a new `StringBuilder` object, you can choose from six different kinds of parameter lists to use:

- ✓ Pass no parameters:

```
Dim sb As New System.text.StringBuilder()
```

- ✓ Pass a single string:

```
Dim s As String = "This"  
Dim sb As New System.text.StringBuilder(s)
```

- ✓ Pass no string, but set aside memory for the specified number of characters (12 in the following example) and permit the string to be enlarged only up to the maximum specified number of characters (44). If you do manipulate this `StringBuilder` in a way that exceeds your specified maximum, an exception (error) is triggered:

```
Dim sb As New System.text.StringBuilder(12,44)
```

- ✓ Pass no string, but set aside memory for the specified number of characters (258 in the following example):

```
Dim sb As New System.text.StringBuilder(258)
```

- ✓ Pass a string and also specify the initial size:

```
Dim sb As New System.text.StringBuilder("Norma Jean",  
258)
```

- ✓ Pass a substring (“cd” in this example), and also specify the initial size:

```
Dim s As String = "abcde"  
Dim startindex As Integer = 2  
'start with b in abcde.  
Dim length As Integer = 2  
'get just bc out of abcde  
Dim capacity As Integer = 14  
' set initial size of sb as 14 characters  
Dim sb As New System.text.StringBuilder(s, startindex,  
length, capacity)  
  
Dim ns As String = sb.ToString  
  
MsgBox(ns)
```

Not only are there those six constructors, but you also find that many of the `StringBuilder`'s methods are heavily overloaded, which means you can pass a variety of different parameter lists to the methods.

The following sections describe a few of the more interesting `StringBuilder` methods.

The Append method

Here's how to use the `StringBuilder`'s `Append` method to concatenate (this is a WebForm example that builds a string for use with HTML code):

```
Sub btnSubmit_OnClick(Sender As Object, E As EventArgs)

    Dim strLabelText As StringBuilder = new StringBuilder()

    strLabelText.Append("Thank you for submitting your
        information.")
    strLabelText.Append("<br /><br />")
    strLabelText.Append("We appreciate your input. ")
    strLabelText.Append("If all our customers were as responsive
        and submissive! ")
    strLabelText.Append("<br /><br />")
    strLabelText.Append("(We don't call it a SUBMIT button for
        nothing!")

End Sub
```

Finally, you use the `ToString` method to transform the `StringBuilder` object into actual text and assign it to your label's `Text` property, like this:

```
lblResponse.Text = strLabelText.ToString()

End Sub
```

Replace and Insert methods

The `StringBuilder` is marvelously flexible. It easily replaces the classic VB `InStr`, `Mid`, and other functions. Here's an example that puts the `StringBuilder` through some of its paces, using a traditional Windows form (WinForm):

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
    As System.EventArgs) Handles MyBase.Load

    Dim s As System.Text.StringBuilder
    s = New System.Text.StringBuilder("My original string.")

End Sub
```

```
s.Replace("My", "This") ' Edit the string by replacing a
                        word.
s.Replace(".", "") ' Edit the string by removing punctuation
s.Append(" has now been extended and modified.") ' append
                        some words
s.Insert(0, "Attention! ", 2) ' insert two copies at the
                        start (character position zero)
MessageBox.Show(s.ToString)

End Sub
```

String\$ Dropped

Instead of the traditional VB `String$` function, you now use the more flexible VB .NET `String` object. It has various methods. For example, use this to create a string with 12 spaces:

```
Dim S As String
S = New String(Chr(32), 12)
MsgBox(S & "!")
```

Tag Property Is Gone

Before VB .NET

Some programmers liked to use the `Tag` property to store information about a control, as a way of identifying controls used in MDI Forms and other highly specialized situations. I never used `Tag`, so I won't miss it.

VB .NET

`Tag` is removed from the language.

TextBox and Clipboard Changes

See “`SelStart` and `SelLength` Are Renamed (Selected Text).”

TextBox Change Event Changes

The `TextBox Change` event has now become the `TextChanged` event.

TextBox Text is Selected

Sometimes when you assign text to a `TextBox`, that text is selected (black on white). To deselect it so that it looks normal, set the `SelectionLength` property to zero.

```
TextBox1.SelectionLength = 0 'turn off the selection
```

Timer Interval Extended

Before VB .NET

The `Timer`'s `Interval` property was limited by the range of numbers that could be held in an unsigned `Integer` data type: 65,535. Because the `Interval` property measures time in milliseconds, this worked out to be a maximum duration of a little over a minute. To use longer intervals, you had to employ a static variable in the `Timer`'s `Timer` event and count up the minutes until that variable reached the number of minutes you wanted.

VB .NET

A `VB .NET` `Timer`'s `Interval` property can range from 0 to 2,147,483,647 milliseconds. (The `Interval` in `VB .NET` is a signed 32-bit `Integer` data type, `Int32`, which can hold a number plus or minus 2,147,483,647.)

In practical terms, this means that you can set the `Interval` to anything between 1 millisecond and 35,791 minutes (which is 596 hours, or almost 25 days). This amount of time should do for most applications.

Timer Tick Event

Before VB .NET

The Timer's `Timer` event was one of the most badly named traditional VB objects. Both the control (the Timer control) and its event (the Timer control's `Timer` event) had the same name. Bad idea. It was always confusing.

VB .NET

Some feel that many of the changes made to traditional VB diction in VB .NET are arbitrary and not helpful (`SetFocus` becomes `Focus`, for example). However, renaming the Timer's `Timer` event to `Tick` is an improvement and was worth changing.

ToolTipText Property Becomes a Control

Many controls in VB 6 had a `ToolTipText` property that you could set to pop up a brief description if the user paused the mouse pointer over the control. In VB .NET, this property is no longer available. However, a `ToolTip` control is on the Toolbox, but at the time of this writing, the `ToolTip` control is not operative.

Top

See “Left, Top, Height, Width, and Move All Change.”

TopLevelControl Property

This new property specifies the top-level control that contains the current control. Usually the form is as follows:

```
MsgBox (Button3.TopLevelControl.ToString)
```

Triggering an Event

Before VB .NET

If you wanted to trigger the code in an event procedure, all you had to do was name that event and its code would execute:

```
CommandButton1_Click()
```

VB .NET

Now, events require two arguments, so you must include these parameters within the parentheses:

```
Button1_Click(sender, e)
```

Literally using `sender` and `e` works just fine. No need to declare these parameters, give them values, or otherwise pay any attention to them.

True Is Still -1

Before VB .NET

The number `-1` was always a fairly weird number to represent `True`, but that's been the VB way since its inception in 1991. `False` has always been `0`, which is intuitive enough, but `True = -1` is strange.

An attempt was made early in the VB .NET design to make VB .NET conform to other .NET languages by making `1` stand for `True`. Thus, when the `Boolean` data type (`true/false`) was converted to other numeric data types, `True` would equal `1` and `False` would equal `0`. However, when any numeric data type was converted to `Boolean`, `0` was to become `False`, but any other number besides `0` was to become `1` (`True`).

This change was abandoned, and the traditional VB `True = -1` was preserved in VB .NET. Presumably this was done for backward compatibility.

VB .NET

True remains -1.

This does not cause any compatibility problems between VB .NET and other .NET languages. When a `True` value is passed from VB .NET to other .NET languages, such as C#, `True` will be passed as 1.

Trim

See “String Functions Are Now Methods” or “LCase and VCase Change.”

Try . . . Catch . . . End Try

See “Error Handling Revised.”

Type . . . End Type Gone

See “User-Defined Type Changes to Structure.”

UCase

See “LCase and VCase Change.”

User-Defined Type Changes to Structure

Before VB .NET

You could use the `Type` command to create a user-defined type:

```
Private Type MyDataType
```

A user-defined type can include a combination of variables of different types (Integer, Long, and so on). This is known as a `struct` in the C language, so can you guess why `Type` is now called `Structure` in VB .NET?

VB .NET

You now declare this entity with the `Structure` command. No fixed-length arrays are allowed. Also, you can no longer use the `LSet` command to copy a variable of one user-defined type to another variable of a different user-defined type. `LSet` is eliminated from the language.

```
Public Structure VariableName
    Dim nas As Integer
End Structure
```

Also note that it was common to use fixed-length strings in user-defined types. Now you cannot do that anymore. You can only use what they call “primitive” data types in a `Structure` — and a fixed-length string isn’t primitive. One workaround is to use the new `PadRight` command (formerly `LSet`) to ensure that a particular string is precisely a particular length:

This is VB 6 code:

```
Type MyType
    RefNo As Integer
    Comment As String*30
End Type
```

To translate that code to VB .NET, use the following code:

```
Structure MyType
    Public RefNo As Integer
    Public Comment As String
End Structure
```

And, because the `Comment` variable is now a variable-length string, if you need it to be a predictable number of characters, you must do that with your programming:

```
Dim strRec As MyType
strRec.Comment = Space$(30) ' set the length
strRec.Comment = "Here is a description."
strRec.Comment.PadRight(30)
```

The `Structure` object is more powerful than the `Type`. A `Structure` can have private members, including both properties and methods. It can have a constructor. A *constructor* is what allows a programmer to specify initial values when instantiating the `Structure` or object. Here’s an example of how a programmer can pass values while instantiating:

```
Dim strNumbers As New Numbers(22, 33)
'create structure & set default values.
```

A Structure is almost a class in flexibility and features (however, limited inheritance and no initializers). Here's an example showing you how to create a simple structure that holds three-column rows (three field records). It also illustrates how to create an array of structures:

```
Private Sub Button1_Click(ByVal sender As System.Object,  
    ByVal e As System.EventArgs) Handles  
    Button1.Click  
    Dim strRecipe(200) As Recipe  
    strRecipe(0).Cuisine = "Mexican"  
    strRecipe(0).Title = "Corn Slump"  
    strRecipe(0).Recipe = "Mix Mexicorn into boiled tamales."  
  
    With strRecipe(1)  
        .Cuisine = "Chinese"  
        .Title = "Egg Drop Soup"  
        .Recipe = "Dribble a beaten egg into 1 qt. chicken stock.  
            Add 1 t. soy sauce. Add 2 T. cornstarch mixed into  
            3 T. cold water. Add 1 t. yellow food coloring."  
    End With  
  
    MsgBox(strRecipe(1).Recipe)  
End Sub  
  
Private Structure Recipe ' similar to a VB 6 Type  
    Public Cuisine As String ' Mexican, Chinese etc.  
    Public Title As String  
    Public Recipe As String  
End Structure
```

This next example shows you how to use private variables and properties:

```
Private Structure Car  
    Public Year As String  
    Private theMake As String  
    Public Property Make() As String  
        Get  
            Make = theMake  
        End Get  
        Set(ByVal Value As String)  
            theMake = Value  
        End Set  
    End Property  
End Structure
```

Finally, the following example shows you how to overload a structure's method and how to use a constructor to permit the specification of initial values when instantiating this structure (see "Overloaded Functions, Properties, and Methods (Overloading)" in this appendix):

```

Private Sub Button1_Click(ByVal sender As System.Object,
                          ByVal e As System.EventArgs) Handles
    Button1.Click

    Dim strNumbers As New Numbers(22, 33)
        'create structure & set default values

    Dim s As String
    s = strNumbers.Larger
    MsgBox(s & " is the largest number now in this structure.")
    s = strNumbers.Larger(2)
    MsgBox(s)

End Sub

Private Structure Numbers
    ' Structure with constructor and method
    Public s As Integer
    Public t As Integer
        ' this constructor will initialize the values when the
        ' Structure is declared:
    Sub New(ByVal InitialS As Integer, ByVal InitialT As Integer)
        s = InitialS
        t = InitialT
    End Sub

    ' Here's the method. It returns the larger
    Function Larger() As String
        If s > t Then
            Return s.ToString
        Else
            Return t.ToString
        End If
    End Function

    ' here's how to overload Larger. This one compares the
    ' submitted string to both S & T
    Function Larger(ByVal NewNumber As Integer) As String
        If NewNumber > s And NewNumber > t Then
            Return NewNumber.ToString & " is larger than s or t."
        Else
            Return NewNumber.ToString & " is smaller than s or t."
        End If
    End Function
End Structure

```

Validate Event Changes

The VB 6 Validate event has now been replaced by Validating and Validated events.

From the Validating event, you can call a routine that validates user input. For example, on a TextBox, you can validate the user input to make sure that it represents the format of an e-mail address. If the text does not validate, you can set `e.Cancel` to `True` (within the validating event) and cancel the event so that the user can fix the entry. The Validated event fires after the control has been validated.

Variable Declaration Changes

Before VB .NET

You declared multiple variables on the same line like this:

```
Dim X As Integer, Y As Integer
```

VB .NET

You can omit repetitive `As` clauses because VB .NET assumes that a list of declared variables all on the same line are the same variable type. The VB .NET equivalent to the previous example is as follows:

```
Dim X, Y As Integer ' (Both X and Y are Integers)
```

Note that if you used `Dim X, Y As Integer` in VB 6, `X` would be a `Variant` type, not an `Integer`. Undefined variables in VB 6 default to the `Variant` type. (There is no `Variant` type in VB .NET.)

Variant Variable Type Is Gone

Before VB .NET

All variables defaulted to the `Variant` variable type. You could, of course, use coercion (“casting”) commands, such as `CBool`, or declare variables `As Integer`, `As String`, or whatever. But if you didn’t, a variable defaulted to the `Variant` type.

VB .NET

VB .NET does not use the `Variant` data type. Each variable inherits from a base object. An object variable type (one declared `As Object`) can hold any data type (just as a `Variant` could).



If the `Option Explicit` is turned off in VB .NET, all variables not declared as a particular type default to the `Object` type.

The `Variant` type, efficient though it often was, had two fatal flaws from the VB .NET designers' perspective. First, in some cases, VB had a hard time figuring out which type the `Variant` should change to — resulting in an error. Second, the other languages in the .NET universe do not use variants — and the .NET philosophy requires conformity between its various languages (at least on the fundamental issues, such as variable typing). Therefore, the `Variant` variable is no longer part of the VB language. It has been banished from VB .NET.

VarType Replaced

The VB 6 `VarType` function, which used to tell you the variable type of a variable (or object), has been replaced by this:

```
obj.GetType.FullName
```

While . . . Wend Changes to While . . . End While

Before VB .NET

VB has always had a looping structure that worked somewhat the same way as the `Do While . . . Loop` structure:

```
While X = 5  
  
'source code that does something  
  
Wend
```

VB .NET

The new `While...End While` structure works exactly the same as the older `While...Wend`; it simply changes the word `Wend` into `End While`:

```
While X < 222
    'source code that does something
End While
```

Width

See “Left, Top, Height, Width, and Move All Change.”

“Windowless” Controls Removed

Before VB .NET

To save processor time and memory, VB programmers sometimes employed label or image controls. These so-called *windowless controls* were drawn on the form instead of being a formal window with a “handle.” Windowless controls take up fewer resources. (Only the label and image were windowless; all other controls are windows.)

VB .NET

The image control has been dropped entirely and the windowless label control is now a fully windowed control. In place of the image control, use the `PictureBox` control.

Unfortunately, the `PictureBox` control doesn’t offer some of the features available via an image control (stretching or shrinking the contained image — which is useful for creating thumbnail views, zooming, and other graphic manipulations). If you’ve written programs relying on these special features of the image control, you have to resort to API calls to get the same effect using `PictureBoxes`.



There are also some other additions to the VB 6 repertoire of controls. Check out the new VB .NET HelpProvider, PrintPreview, ErrorProvider, PageSetupDialog, CrystalReportViewer, LinkLabel, Button (formerly known as CommandButton), MainMenu, RadioButton (formerly OptionButton), DomainUpDown, NumericUpDown, and ContextMenu.

Zero Index Controversy

We humans almost always count from 1. It's natural to our way of describing, and therefore thinking, about numbers and groups. When the first person arrives at your party, you don't say "Welcome, you're the zeroth one here!" And when your child is one year old, you don't send out invitations titled "Jimmy's Zeroth Birthday Party!!"

It makes little logical or linguistic sense to humans to begin counting from zero when working with lists, collections, or indeed arrays. We quite properly think of zero as meaning nothing — absence, nonexistence.

However, it was decided early in the development of computer languages that it would be more "computer-like" if we started indexing arrays (and some other collections) using zero as the starting number: `Array(0)`, `Array(1)`, and so on. In fact, a zero-based counting system is more computer-like, but that's no compelling argument for using it in a computer language, which is a tool created for use by people. There are technical reasons why a zero-based counting system was used to save computer memory — but this reason has long since lost any validity.

Many programmers — having learned to adjust to zero-based lists — want to perpetuate this unfortunate practice. Zero-based lists have resulted in countless man-years of debugging problems (mismatching loops to upper array boundaries — should it be `For I = 1 to 10` or `For I = 1 to 9`? — and such). And it continues to this day. Some sets in computer languages start their index with 1 (some but not all collections of objects, for instance) while other sets start with 0 (traditional arrays).

Before VB .NET

Visual Basic improved on this counterintuitive mix-up by permitting programmers to use the `Option Base 1` command, which forced all arrays to begin sensibly, using an index of 1 as their first element. (You could also force an initial index of 1 by this kind of array declaration: `Dim MyArray(1 To 100)`).

VB .NET

The `Option Base 1` command has been deleted from the language and can no longer be used. All arrays must now begin with 0. Also, you can no longer use the `(1 To 100)` format when dimensioning an array to make the array be one-based. What's more, VB has traditionally counted characters using one-based lists. No longer.

This VB 6 function returns ABC because when you specify that you want the string starting at the first character `(, 1)`, you mean the first character — A in this case:

```
Dim x As String
x = Mid("ABC", 1)
MsgBox(x)
```

When you run this example, you get ABC as you would expect.

Now, what do you suppose happens when you use the equivalent VB .NET function `Substring` and specify 1 as the first character?

```
Dim x As String = "ABC"
MsgBox(x.Substring(1))
```

Perhaps you're not stunned, but I was. When I first tried this example, I thought I was saying to VB .NET that I wanted a string starting at the first character `(1)`, which would give me back ABC. Nope. You get BC! In VB .NET, this `(1)` means `(2)`. The string that is returned to you starts with the second character, even though you used `(1)` in the source code. This phenomenon, they tell me, is deliberate, by design, and in some way good. How it's good escapes me.

It should be noted that in VB .NET, `Mid("ABC", 1)` still returns ABC.

I expect that when psychoergonomics are, at long last, introduced into computer-language design, we'll get rid of inane, human-unfriendly usages like the zero-based set. There is simply no reason at all to transfer to us humans the burden of describing some sets in computer programming as beginning with 1 and other sets as beginning with 0. All groups should begin with 1, obviously. Why should we memorize exceptions to a rule that is a very bad rule in the first place?

Excerpted from Visual Basic® .NET All-In-One Desk Reference For Dummies® by Richard Mansfield. Copyright (c) 2003 by Wiley Publishing, Inc. All rights reserved.